Acta Informatica 29, 241–265 (1992)

The *SB*-tree
An index-sequential structure for high-performance sequential access

Patrick E. O'Neil

Department of Mathematics and Computer Science, University of Massachusetts at Boston, Boston, MA 02125-3393, USA

Received June 7, 1990 / November 18, 1991

Abstract. A variant of a *B*-tree known as an *SB*-tree is introduced, with the object of offering high-performance sequential disk access for long range retrievals. The key to this efficiency is a structure that supports multi-page reads (or writes) during sequential access to any node level below the root, even following significant node splitting. In addition, the *SB*-tree will support a policy to 'stripe' successive multi-page blocks on multiple disks to achieve maximum parallelism. Compared to traditional *B*-tree structures, *SB*-tree performance characteristics are less subject to degradation resulting from modifications entailed in growing and shrinking; *SB*-trees are therefore more appropriate for use in situations where frequent reorganization is not possible. A performance analysis reveals the strengths of the *SB*-tree by comparing its performance under various circumstances to the B^+-tree and the bounded disorder (BD) file of [11]. The performance analysis formulates a new useful concept, the 'effective depth' of an *SB*- or B^+-tree, defined as the expected number of pages read from disk to perform a random retrieval search given standard buffering behavior. A graph of effective depth against tree size is shown to have a scalloped appearance, reflecting the changing effectiveness of incremental additions to buffer space.

1. Introduction

It is the aim of this paper to introduce a variant of the *B*-tree structure of Bayer and McCreight [2, 3]. The structure introduced supports multi-page disk read/write access for long sequential range retrievals and updates following significant node splitting, and we refer to it as an *SB*-tree (the "*S*" in "*SB*-" stands for "Sequentially Efficient"). The terminology used in this paper is taken from [6, 11, 17].

The efficiency that can be achieved in reading a contiguous multi-page block rather than many single pages is well known [6, 8, 11, 15, 16]. To quote from [16], Chapter 3, speaking of the characteristics of an IBM 3380 Disk, "... The time to complete a [read of a single page] could be estimated to be about

20 ms (assumes 10 ms seek, 8.3 ms rotational delay. 1.7 ms read) .... The time to perform a sequential prefetch read [of 64 contiguous pages] could be estimated to be about 125 ms (assumes 10 ms seek, 8.3 ms rotational delay, 106.9 ms read of 64 records [pages]), or about 2 ms per page." Other researches break down the disk startup time, time needed to bring the disk arm to the right position to begin the read, into somewhat more precise categories, including an initial "Protocol time" [7], and a final "Settle time" [8], etc. However, it is unquestioned that the average time to fetch a page can improve by a factor of ten or more when a large number of needed pages are batched contiguously to share startup time.

Applications which require frequent long range retrievals entailing reads of hundreds of pages in sequence are quite common in commercial applications. As an example, households in direct marketing databases are often clustered by zip-code and street address to reduce processing time for mailings to prospects chosen by location (see [12]). Multi-page block reads are currently supported in a number of commercial database products (INGRES [10]. DB2 [14]), with significant query performance enhancement, as shown for example in measurements of [13]. With current *B*-tree structures, however, the multi-page read strategy for these products is a passive one. Successive leaf pages of a *B*-tree which happen to lie on successive disk pages will be accessed in a contiguous block, and *B*-tree load utilities are careful to keep this logical/physical correspondence intact. But as new information is added and *B*-tree node splits occur, the logical/physical correspondence of contiguous pages quickly breaks down, with a resulting degradation of performance. The purpose of the *SB*-tree is to support multi-page block reads for long range-retrievals at all node levels below the root following significant node splitting activity, while still maintaining single-page node access efficiency during exact match retrieval.

The *SB*-tree, then, has the following desirable properties:

(1) Multiple nodes in key sequence order at any level of the tree below the root can be read (or written) with a single multi-page disk access, thus supporting high performance range-retrieval.

(2) Accesses to single records (on a leaf node) can be performed with a single page access at each node level in the access path.

(3) Ultra high-speed sequential access where successive long blocks of pages are "striped" onto different disks can be supported under the *SB*-tree (e.g. see [5]).

(4) The high performance characteristics of an *SB*-tree are much less subject to degradation than traditional *B*-tree structures when the tree is modified to accommodate changing information. For this reason, it is appropriate for situations where regular *B*-tree reorganization cannot easily be performed.

In Sect. 2 following, we present the basic structure of the *SB*-tree. In Sect. 3 we compare its efficiency with various competing structures. Section 4 comments on other considerations of importance. Section 5 contains our conclusions.

2. The *SB*-tree structure

The *SB*-tree requires for its creation and maintenance a pool of "multi-page storage blocks". Each block consists of a set of L contiguous pages on disk

(which can be read or written with a single sweep of the disk arm). As an example, we might take $L=64$. From the first page encountered by the disk head in reading or writing a storage block to the last one in the block, the pages are numbered 1, 2, ..., L.

2.1 Growing and shrinking the SB-tree

The *SB*-tree has the hierarchical (single-page) node structure of a B^+-tree or B^*-free [6]. In both B^+-trees and B^*-trees, the records and the keys identifying them are stored only in the leaf nodes, and a sequential scan can be performed by following sibling pointers between adjoining leaf nodes. (Note that our leaf level entries can be either file records clustered by primary key, which we assume in what follows, or secondary index entries with pointers to file records, clustered by secondary key. As explained later, we will not be using sibling pointer access in the *SB*-tree.) A B^*-tree has the same structure as a B^+-tree, but modifies the insertion algorithm by an "overflow" step as follows. When an insert causes a leaf to become overfull, before carrying out a split, we inspect the neighboring sibling nodes; if space exists, we reorganize data onto two buckets so that a split is avoided; otherwise we take a full sibling leaf node and, together with the one receiving an insert, reorganize into three leaf nodes so that each one contains about 2/3 of the information; we refer to this as a "2–3 split". We continue to use the term "*B*-tree" loosely, to refer to the B^+-tree and B^*-tree structures.

In what follows, we propose a basic definition for the *SB*-tree which avoids the overflow step and 2–3 split. In Sect. 4 of [17], it is pointed out that the survival interval of a B^*-tree node (average number of inserts and deletes between merges and splits) is thereby reduced; indeed, it may be for this reason that up until now B^*-trees have been avoided in commercial products. However, we point out in our own analysis of Sect. 4 that multi-node (three node or more) overflow and underflow reorganizations may be appropriate in cases where the frequency of read accesses (and especially of range searches) greatly exceeds the frequency of update activity. In that case, the gain from placing more information on fewer pages more than makes up for the (infrequent) extra cost of reorganization.

For now, we shall follow [17] in defining *SB*-tree operations like those analyzed for a *B*-tree, to provide "distance" from the triggering condition for splits, borrows, and merge steps, and thus guard against frequent "chattering" near the boundaries. Assuming fixed-length entries, and an absolute upper bound of b entries in a node, we say that the number of entries in the node is called the "stage" of the node.

Definition 2.1 The *borrow just enough B*-tree policy of Zhang and Hsu is parameterized by the choice of two values, β and f, where β is an integer, $0 \leqq \beta \leqq (b+1)/2$, and $0 \leqq f \leqq 1$, $f\beta$ an integer. The policy is described as follows:

(1) The stage of a node ranges between a (the lower bound) and b (the upper bound).

(2) $a=((b+1)/2-\beta) \geqq 0$, where β is an integer, $0 \leqq \beta \leqq (b+1)/2$.

(3) Insertion into a node in stage b causes the node to split into two nodes of stage $(b+1)/2$.

(4) Deletion from a node in stage a causes the node to (i) borrow entries from a neighbor to bring about equal fullness (following the analysis in [17], Sect. 4), if after borrowing both nodes are at least $a+f\beta$ full, and (ii) merge with the neighbor if after the borrow attempt one of the nodes would be less than $a+f\beta$ full. For equally likely inserts and deletes, the parameter f turns out to have an optimal value of 2/3.

Let us now begin to define the *SB*-tree by tracing its structure during early growth. When the first entry of an *SB*-tree is inserted, a single page is allocated for the first node of the *SB*-tree. This node is a root node and leaf node at once, as is the case also with the B^+-tree. Successive entries are added to this node until an insert forces a split in the root node. This node is then split into two leaf node pages which occupy page 1 and page 2 of a multi-page block (see Fig. 1). The root node, now containing a single separator and two offspring pointers remains on the original page allocated; thus we avoid needing to modify external structures which point to the root. Note that the structure created so far is simply a B^+-tree of depth two, where the two leaf pages sit adjacent to each other on disk. A small amount of extra information is held in directory nodes of the *SB*-tree, to contain information about multi-page blocks of children nodes, as will be explained later.

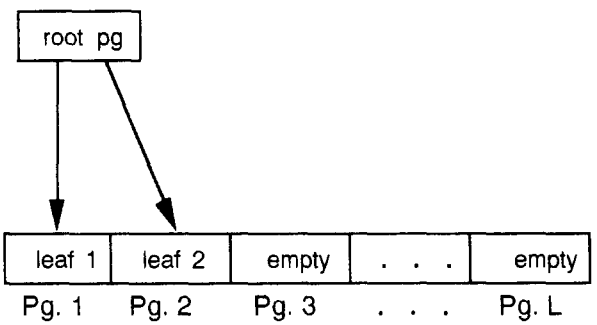

Fig. 1. An *SB*-tree with two leaf nodes on a multi-page block

At this point let us inductively assume that we have arrived at an *SB*-tree of multiple nodes with depth two or more. We have the following definition.

Definition 2.2 INDUCTION HYPOTHESIS FOR THE *SB*-TREE
In addition to the four properties in Definition 2.1, the following properties connect *SB*-tree nodes with multi-page blocks.

(5) Every node other than the root node of the *SB*-tree sits on a multi-page block. The root node does not, and therefore the next two properties do not apply.

(6) Every multi-page block containing a node from the *SB*-tree contains a sequence of K nodes, falling in the initial K pages of the block, such that the following property is maintained: if any two nodes, A and B fall on the same multi-page block R, then A and B are on the same level of the tree and all intervening nodes in key sequence between A and B also must fall on the same

multi-page block R. We do not require that the nodes appear in key order on the block.

(7) We draw an analogy between the K nodes of a multi-page block and the entries of a B-tree node: an analogous policy of "block-splits", "block-merges" and "block-borrows" is pursued, so that the number of nodes, K, on a multi-page block with L pages falls in the range: $a' \leqq K \leqq b'$ (where $b' = L$), for any level where at least a' nodes exist. Completing the analogy, we have the parameters β' and f', where $a' = (b'+1)/2 - \beta'$, β' an integer, and $0 \leqq \beta' \leqq (b'+1)/2$, $0 \leqq f \leqq 1$, $f'\beta'$ an integer. Details of block actions follow.

Note that the properties of this induction hypothesis can be thought of as holding for a tree consisting of a single root node as well as the three-node tree of Fig. 1. In both of these examples, the number of nodes at the leaf level is too small for the lower bound, a', of property (7) to be relevant.

Consider Fig. 2. A node at level $J-1$ (level 0 is the root) contains offspring page pointers $(P_1, P_2, \ldots)$ and intervening key separators $(S_1, S_2, \ldots)$. Ignore, for the moment, the entry M in the node entry structure. This contiguous sequence of page numbers, P_1 through P_K, points to nodes at level J, node 1 through node K, which sit on the initial K pages of a single multi-page block. We note that the nodes need not be in increasing order by key on the multi-page block, but Property 6 does not permit empty pages to intervene between node pages. Furthermore, if two nodes on level J lie on the same multi-page block, then so do all intervening nodes in key sequence.

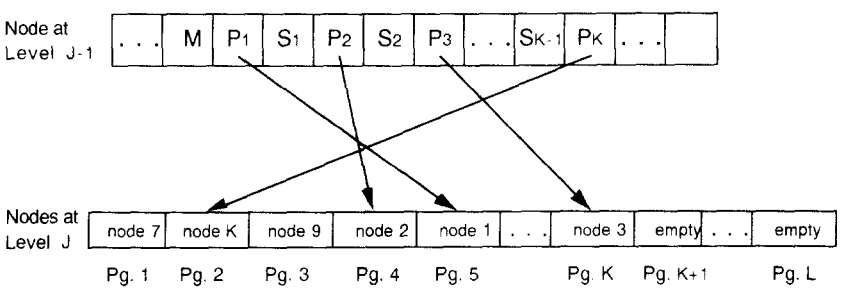

Fig. 2. A contiguous key sequence of nodes on a multi-page block

We now consider updates to the *SB*-tree and indicate the manner in which the *SB*-tree structure is modified so as to maintain the appropriate node structure in relation to multi-page blocks.

2.1.1 Insertion. As new entries are inserted at the leaf level of the *SB*-tree, the number of entries increases until some insert causes a leaf to become overfull and a node split occurs. Whenever a node split occurs at any level of the tree, the *SB*--tree algorithm will look for the first empty page on the multi-page block containing the node receiving the insert. As we see from Fig. 2, this will be page number $K+1$ in the containing multi-page block, where K nodes already exist; sufficient information will be kept at the directory level above the splitting node so that this empty page in the block can be easily found if space remains. If an empty page exists, we place the new node created by the split on that

page. (The root page, with no higher level directory information, receives special handling. A new multi-page block is allocated and the two new root children placed on it while the root page remains in place, as in the split from one to three nodes explained earlier.)

Following the split, we see that properties (5) and (6) and (7) of the induction hypothesis still hold. The only property requiring some thought is (6): we see that the contiguous key sequence of the two nodes resulting from the split is equivalent to that of the single node which just split, and thus, by the induction hypothesis, the desired property is inherited.

If no empty page of the multi-page block exists, then a "block-split" operation is necessary. A new multi-page block, S, is allocated from the pool, and the overfull multi-page block, R, containing the splitting node is read into memory with a multi-page read. Then the $L+1$ (or $b'+1$) nodes of the block R are disposed as follows: the initial contiguous key sequence of $L/2+1$ nodes remain in place on block R and the remaining contiguous key sequence of $L/2$ nodes are reallocated to the initial positions of block S. Then an operation called "collapse-left" is performed on the $L/2+1$ nodes on block R, moving the minimum number of nodes so that all of them will fall on the initial $L/2+1$ pages of the block. (All nodes which sit originally on this initial set of pages remain in place, and other pages are moved once to fill the gaps in the set.) Note that only one node split takes place, and new directory information associated with this node split can recursively cause splits in the ancestor nodes. The movement of the other nodes from block R to block S and within block R will simply cause the corresponding page numbers of the parent index entries to be modified, but with no potential for a split. Clearly the block-split operation is defined so as to result in a structure which obeys properties (5) and (6) and (7) of the SB-tree Induction Hypothesis.

One further detail of the block-split operation remains to be described. We have been assuming that the structure of the *SB*-tree index nodes is similar to that of a B^+-tree. However, to most efficiently determine the boundaries of multi-page blocks, we use a special "marker entry", M, in the index. This marker contains the disk page number where a multi-page block begins, and the current length of the string, K, and it is stored as an entry in the directory structure immediately preceding the string of entries for a block of nodes below. We are assuming then that the directory structure accommodates entries of three types: separators, S_i, subsidiary node pointers, P_i, and markers, M. See again the example of Fig. 2, where the significance of the entry M is now clear. A block-split will cause a new marker entry to be inserted in the directory, and therefore has the potential to cause a directory node split, even when new separator entry for a node split did not.

In the case where a string of entries for a block falls on two different pages on the directory level, the marker M for the string is replicated so that it falls also at the beginning of the remaining string fragment on the continuation page. In addition, each index node header contains a pair of flags. CONTINUEDFROM and CONTINUEDTO. The flag CONTINUEDFROM, is set to YES if the left-most entry on the index node is a marker for a continued block string of entries from the left sibling index node; it is otherwise NO. The flag CONTINUEDTO has a similar significance for a block string being continued to the right-hand sibling. The maintenance of these flags is a relatively simple matter during operations of index node splitting, borrowing and merging, and

the analogous operations for blocks. The algorithms for these operations should obviously show some preference for a string of entries for a block lying entirely on one index node, but how strong this preference should be depends on specific parameters of the file design, such as $L(=b')$, β', and f'.

The value of the marker entry, M, should be evident. Whenever a node split becomes necessary, we search entries at the directory level to the left of the entry for the split node until the marker entry is located (a back-pointer from the entry to the marker seems counter-productive in terms of index space utilization). The marker entry tells us what page is free in the containing multi-page block: we have the initial page number, I, of the block, and the number of pages, K, used so far, so the next page number can be calculated as $I+K+1$. We must then update the number of pages used in the marker M, and also in any replicated marker on a sibling index node if either CONTINUEDFROM or CONTINUEDTO are set. In case the number of used pages K on the marker is equal to L, we will see that a block split is necessary, and during the block split we will create a new marker.

We do not propose to maintain sibling pointers between nodes at any level of the *SB*-tree. Each directory marker entry, M, provides us with the number of initial pages K *on each subsidiary block*, and, as we see in the following section, we thus obtain a substantial performance advantage both in terms of *I/O* duration and buffer space utilization during range retrieval. We therefore assume that the range operation also reads in sequences of directory pages (or blocks) in order to access the relevant marker entries. While we could replicate the value K in a sibling pointer with a certain saving of *I/O* during long range retrieval, there would be added complication and *I/O* in keeping these values up to date during splits and merges. The idea of sibling pointers is not out of the question for low update situations, but it is of marginal use and therefore not included in our basic design.

2.1.2 Deletion of SB-tree entries. When an entry is deleted from a leaf node, we follow the normal B^+-tree policies of node borrowing and node merging, as explained in Definition 2.1, part (4). Note that it is not necessary that the neighboring nodes being merged belong to the same multi-page block, but for simplicity of describing this algorithm, we shall assume they are. If the parent entries pointing to the nodes being merged fall on different index nodes, then we have flexibility in choosing the node to become empty, preferring that separator entries for nodes lying on a multi-page block should lie entirely on a single directory node. Node merging may also lead us recursively to perform borrowing or merging on the parental index nodes.

When two nodes are merged which sit on a multi-page block with K active pages, a "hole" may be left on page i, $i<K$. If this happens, then the node on page K must be moved to fill in page i, so that the remaining pages fall in the initial $K-1$ pages of the block as required by property (6) of Definition 2. However, if the number of pages remaining falls to a point such that $(K-1)<a'$, then the system will perform a block-borrow of nodes from a neighbor (moving those adjoining nodes in key sequence order into the current under-full block to equalize the number of nodes per block, and correcting the parental index pointers) if, after borrowing, both blocks contain at least $a'+f'\beta'$ nodes; if the borrow attempt does not fulfill this condition, a block-merge will be performed: all the nodes on two adjoining multi-page blocks will be moved to

occupy a single one of the blocks, and the parental index pointers of the nodes moved will be corrected.

Note that the block-borrow operation does not free up any disk storage blocks, so we look for its motivation in other areas. A block-merge is an operation we want to perform, since this will improve the efficiency of long range retrievals in the region, and we clearly require some "trigger" to evaluate the advisability of a block-merge. Low usage on some block is the most appropriate trigger to evaluate block-merge, but we wish to avoid the cost of frequent inappropriate evaluations, so we perform the block-borrow operation to bring the low node occupancy to a higher level and avoid frequent triggers.

3. Performance evaluation of the *SB*-tree

In this section, we will compare the cost of operations in an *SB*-tree with the cost of the same operations in comparable data structures. We start by comparing the *SB*-tree to the B^+-tree; later we compare performance of the *SB*-tree with that of the bounded disorder file (BD file) of [11].

3.1. Comparison with B^+-tree performance

We use the following notation for parameters of a B^+-tree or an *SB*-tree and the system referencing them.

S – Size of index separators (average), in bytes. (Example, 8 bytes)
R – Record (entry) size in bytes, leaf level. (Example, 200 bytes)
P – Disk page size, in bytes. (Example, 4 K bytes)
D – Depth of the tree.
F_{leaf} – Average fractional "fullness" of a leaf node. (E.g. 0.70)
F_{node} – Average fractional "fullness" of an interior node. (E.g. 0.70)
B – Buffer size in disk pages (tree nodes). (Example, 1000 pages)

3.1.1 The "effective depth" of a B^+-tree. In order to achieve realistic performance estimates, we need to evaluate B^+-tree behavior in a page buffered system. It is well known that the root node is usually found in memory buffers, but this is not the whole story. We offer the following rough analysis to estimate the "Effective Depth", D_E, of the B^+-tree, which we define as the expected number of pages *not* found in memory buffers during a search from the root to the leaf for an entry with a random key value. We assume for the analysis that there are B disk pages in memory buffers at any time, and that these buffers contain the most recently accessed B nodes of the tree (LRU algorithm).

The maximum fanout of an index node is about $P/(S+4)$, where we assume the separator and page pointer for each offspring node together take up $(S+4)$ bytes. A typical value for maximum fanout is 340, where separators are assumed to be eight bytes in length and a page is 4 K bytes with a small number used for overhead. The *average* fullness F_n of an internal B^+-tree node following many random changes has been shown to be about 70% ([1], Table 5.1.), result-

ing in a fanout of about 238. Because the root node fanout can vary anywhere from 2 to $P/(S+4)$, and this is a rather unstable function of the total number of entries, we must look for some characteristic level of the B^+-tree which is more useful for our analysis.

Let level W of the tree be the lowest level containing a number, Z, of nodes where we may roughly assume that all of the nodes on the levels above level W are consistently found in memory buffers because of their frequency of reference (and indeed that they take up insignificant space), and that only a small fraction of nodes on levels below level W are found in buffers. Then if D' is the number of levels below level W, the "Effective Depth" D_E of the B^+-tree will in the range: $D' < D_E < D'+1$. For a more accurate estimate, we need to analyze the percentage of nodes on level W which will lie in buffers. We assume that the number B of buffer pages has a value of at least several hundred (an assumption which becomes more realistic, even with multiple $B+$-trees in use, as memory prices decrease with time), so that W is not the root level or even the level just below the root if that contains only a few nodes.

To start our analysis of buffer occupancy on level W, picture the set of nodes that are present in buffers as follows. We say that the "trace" of a search is the set of nodes on level W and below that are accessed during a search from the root to the leaf. In an LRU scheme, all the pages of the most recent n traces will be in buffer, where n is some function of B and the tree structure to be determined. The number of pages at level W and below in a given trace is $D'+1$, and we take a first approximation to the number n of recent traces in buffer by setting $n = B/(D'+1)$. (To reiterate, we assume that nodes above level W are constantly in buffers, and require insignificant space). This approximation assumes that the traces don't intersect on nodes of the tree, a reasonable approximation for nodes below level W, but requiring later correction for the set of nodes at level W itself.

From the Z nodes on level W, we now estimate the expected number of nodes that are NOT in one of the most recent $n = B/(D'+1)$ traces. Assuming that the most recent n traces intersect nodes at level W in an independent random fashion, the probability P_0 that any given node n_0 on level W is not in any of the most recent n traces is the probability that n nodes in sequence were selected from among the other $Z-1$ nodes:

$$P_0 = (1-1/Z)^n.$$

The expected number of nodes on level W that are not in buffer, E_0, is $Z \cdot P_0$, and using the asymptotic approximation $(1-1/Z)^Z \approx e^{-1}$, we see that:

$$E_0 \approx Z \cdot e^{-n/Z}, \text{ or, } E_0 \approx Z \cdot e^{-B/(Z(D'+1))}.$$

With this formula, we can improve our estimate of n, by replacing "1" in the formula $n = B/(D'+1)$ by $G = E_0/Z \approx e^{-B/(Z(D'+1))}$, the fraction of nodes on level W which are not in buffer. This gives a new approximate value for E_0, $E_0 \approx Z \cdot e^{-B/(Z(D'+G))}$. The feedback correction for G can be performed recursively (G cannot be solved in closed form), but one recursion is usually sufficient for reasonable accuracy. By the definition of G, the value of the effective depth of the tree is given by:

$$D_E = D' + G.$$

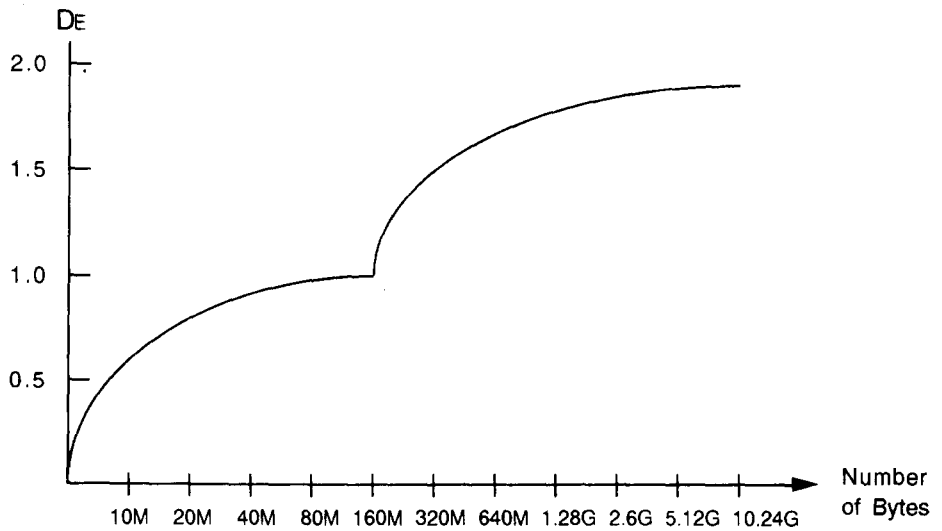

Fig. 3. D_E, the effective depth of a B^+-tree vs the number of bytes at the leaf level

Figure 3 graphs the effective depth of a B^+-tree against the number of bytes at the leaf level, assuming the number of buffers B is 1000, leaf nodes and internal nodes are 70% full, and index separators are 8 bytes long. In a few cases two different levels for W are tried and the results averaged. The D_E values from which Fig. 3 is graphed are tabulated as follows. The first line below corresponds to the number of bytes at leaf level, and the second line to the effective depth, D_E.

Number Bytes:										
10 *M*	20 *M*	40 *M*	80 *M*	160 *M*	320 *M*	640 *M*	1.28 *G*	2.56 *G*	5.12 *G*	10.24 *G*
Depth, D_E:										
0.69	0.87	0.93	0.98	1.02	1.21	1.52	1.74	1.87	1.94	1.97

It is interesting to note that the graph of Fig. 3 presents a scalloped appearance. This indicates that increasing the ratio of memory buffer size to B^+-tree size has an important effect in certain ranges, where a large proportion of a B-tree level can be contained in memory, but that this effect is of diminished importance in other ranges, where the proportion of the level contained becomes less useful in saving real *I/O*s.

3.1.2 Random retrieval. We can now compare the cost of specific operations on the B^+-tree and the *SB*-tree, starting with the time to search for a random key-value entry in the tree. In doing so, we try to define a consistent and suggestive notation for resource costs, disk accesses (DA_{task}) and CPU time (CPU_{task}), associated with each task we encounter. Our analysis results in formulas for resource use for each particular type of access, in terms of resources for more basic tasks which are more intuitive in their magnitude.

The number of disk accesses needed to locate a specific entry on the leaf level by key, DA_{random}, has expected value equal to D_E, the effective depth of the tree as derived in Sect. 3.1.1. We note that the UNITS of all DA_{task} measures are in elapsed time and are measured by the number of single page

$$DA_{random} = D_E$$

$$CPU_{random} = (D\text{-}1)\cdot CPU_{nsearch} + CPU_{lsearch} + D_E\cdot CPU_{pgovhd}$$

Fig. 4. DA and CPU costs for random retrieval in a B^+ or *SB*-tree

accesses; a single page access may be thought of as requiring 20 ms. The CPU time required is given by, $CPU_{random}=(D-1)\cdot CPU_{nsearch}+CPU_{lsearch}+D_E\cdot CPU_{pgovhd}$, where D is the tree depth, $CPU_{nsearch}$ is the average CPU time needed to search a single index node, and $CPU_{lsearch}$ is the CPU time needed to search a leaf node. The term given by CPU_{pgovhd} represents the CPU overhead required for each single page of disk access (e.g. to set up a channel program). See Fig. 4.

The only difference between a B^+-tree and an *SB*-tree in this case is that the *SB*-tree contains entries of type M in its index nodes, and therefore the tree is larger by a very slight factor. However, it can be shown that the effective depth D_E exhibits smooth behavior even at the point where a split occurs and the depth D is incremented by 1. For this reason we treat the two tree structures as having identical random retrieval cost.

3.1.3 Range-retrieval. A range-retrieval operation retrieves m records (or secondary index entries) in sequence from the leaf level of the tree, with key value in the range (k_s, k_e). The operation starts with a search from the root to the leaf level for the smallest key-value entry equal to or exceeding k_s; thus we entail the CPU and DA cost of a random retrieval. Following this, successive entries at the leaf level are accessed (with CPU cost of CPU_{next}) until the key value for some entry exceeds k_e. To estimate the expected number of leaf nodes which are read during a range retrieval of m records, we apply Lemma 3.1.

Lemma 3.1 *See Fig. 5 below. A segmented line is ruled in inches, with every* k^{th} *rule marked to start a line segment k inches wide. A straight stick that has an integer length of r inches lies along the segmented line at a random inch-ruled starting position. Then this stick will intersect n of the line segments at more than a single end-point, with the expected value of n, E(n), given by:* $E(n)=1+(r-1)/k$.

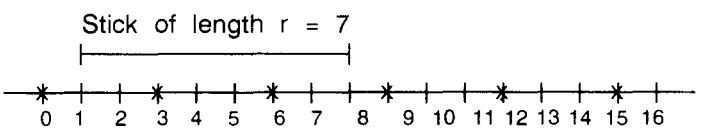

Fig. 5. On an inch-ruled line with markings $k=3$ inches apart, a stick of length $r=2k+1$ will cross exactly 2 marks

Proof. A stick of length $r=m\cdot k+1$ will always cross exactly m marks to lie on $n=m+1=1+(r-1)/k$ segments at more than a single end-point. Consider now a stick of length $r=m\cdot k+s+1$, with $0\leqq s<k$. This will lie on $m+2$, rather than $m+1$ line segments, exactly when the length from the starting position of the stick to the first mark is v, $0<v\leqq s$. Thus the probability of $m+2$ line segments is s/k, and the expected number of line segments covered is:

$$(s/k)(m+2)+((k-s)/k)(m+1)=m+1+s/k=1+(r-1)/k. \quad \square$$

The average number of records on a leaf page, with the average leaf fullness given by F_{leaf}, is $P \cdot F_{leaf}/R$. (See the notation at the beginning of Sect. 3.1.) Given this average, which corresponds to the k of Lemma 3.1, the number of leaf pages intersecting a sequence of m records is $1+(m-1) \cdot R/(P \cdot F_{leaf})$. From this, we can derive the disk access and CPU values for a range retrieval on a B^+-tree, given in Fig. 6. Note that the first of the pages in range will be brought into memory in the course of searching from the root to the leaf; the DA overhead for this I/O is already in D_E and the CPU overhead is already included in CPU_{random}.

$$DA_{brange} = D_E + (m\text{-}1) \cdot R/(P \cdot F_{leaf})$$

$$CPU_{brange} = CPU_{random} + m \cdot CPU_{next} + (DA_{brange} - 1) \cdot CPU_{pgovhd}$$

Fig. 6. B^+-tree DA and CPU costs for range retrieval of m entries

Performance of long range retrievals in an SB-tree is much improved over a B^+-tree, as intended, because we are always able to perform multi-page reads to retrieve successive leaf pages (indeed, multi-page reads at index node levels are also possible, and may be of some value in very long ranges.) Information in the directory allows us to determine what multi-page blocks below should be read: starting with k_s, we pass over successive separators, reading in all multi-page blocks when more than four or five node pages from a block are accessed, until one of these separators exceeds k_e. The initial "M" type entry in the directory tells us the initial page and number K of pages in each multi-page block. Note that since the nodes to be retrieved always lie on the containing multi-page block in the initial K pages, we can perform a disk access which retrieves only the first K out of L pages of the block, and often saves significant elasped time. We designate the measure of elapsed time for a disk access of this kind by $DA_{long(K)}$, measured in units where a single page access takes elapsed time 1; $DA_{long(K)}$ has value of about 4.5 for common blocks where $K = 0.7 \cdot L$, as estimated in the second paragraph following.

$$DA_{sbrange} = D_E + (((m\text{-}1) \cdot R/(P \cdot F_{leaf}))/(L \cdot F_{bleaf})) \cdot DA_{long(L \cdot F_{bleaf})}$$

$$CPU_{sbrange} = CPU_{random} + m \cdot CPU_{next} + D_E \cdot CPU_{pgovhd} + (DA_{sbrange} - D_E) \cdot CPU_{lgovhd}$$

Fig. 7. SB-tree DA and CPU costs for multi-page range-retrieval

The expected number of containing multi-page blocks for a sequence of m records on $1+(m-1) \cdot R/(P \cdot F_{leaf})$ pages, where the multi-page blocks contain an average of $L \cdot F_{bleaf}$ pages, can also be calculated using Lemma 3.1, with the result we see in Fig. 7 as a factor of $DA_{long(L \cdot F_{bleaf})}$. In these formulas, we ignore situations where single page accesses might be superior. In particular, short range accesses involving only a few pages of an SB-tree will have the same

costs as those of the B^+-tree, as formulated in Fig. 6. We also ignore the relatively insignificant *I/O* required for long range retrievals to keep superior directory nodes in buffer to make up for the lack of sibling pointers.

Comparing the elapsed time for *I/O* required for disk access in an *SB*-tree ($DA_{sbrange}$ in Fig. 7) with that required in a B^+-tree (DA_{brange} in Fig. 6), we find the improvement for which the *SB*-tree was designed. For long ranges, where the D_E term is swamped, the ratio of elapsed time for an *SB*-tree to that for a B^+-tree is proportional to $DA_{long(L \cdot F_{bleaf})}$ and inversely proportional to $(L \cdot F_{bleaf})$. To see what this means, we assume that $L=64$, $F_{bleaf}=0.70$, and use the *I/O* time figures of [16] quoted in Sect. 1: a single disk access (our unit of 1 for elapsed time) requires 20 ms, and $DA_{long(0.7L)}$ represents 93.5 ms (10 ms seek, 8.3 ms rotational delay, and 75.2 ms to read 45 pages). Thus $DA_{long(0.7L)}=93.5/20$, the ratio of elapsed time for a multi-page block access to a single disk access, or about 4.5 as mentioned earlier. Dividing by the average number of pages accessed in a multi-page block, $(0.7 \cdot L)$, about 45, we see that our *SB*-tree speedup is a result of reading 45 pages in 93.5 ms instead of $45 \cdot 20\ \text{ms} = 900$ ms, an elapsed time ratio of 0.1039, representing a 10 to 1 speedup.

The CPU usage for a long range retrieval is also significantly reduced in the *SB*-tree. The CPU_{random} term will be swamped by a long retrieval, and the $m \cdot CPU_{next}$ term should be quite small. The CPU overhead for reading a multi-page block (CPU_{lgovhd}) is only two or three times greater than for reading a single page (CPU_{pgovhd}), and thus we see the same order of improvement for and *SB*-tree over a B^+-tree in CPU time as we did in disk accesses.

3.1.4 Insertion. Insertion of a new entry into a B^+-tree entails, at a minimum, searching from the root for the proper leaf location, placing the new entry in the leaf node, and writing back the leaf node modified. So far, we have:

$$DA_{binsert} = DA_{random} + 1; \quad CPU_{binsert} = CPU_{random} + CPU_{linsert} + CPU_{pgovhd}.$$

With probability P_{lsplit}, the leaf node will have no room for the new entry, and a split will occur. In this case, we must copy half the entries of the splitting node into a new buffer page (CPU_{lsplit}) and write the newly created sibling node out to disk (a page write); in addition, we must correct the sibling pointer of some sibling node to the leaf being split, a page read and write, and write the parent node out with a new separator. At this point, the expected value for $DA_{binsert}$ is $DA_{random} + 1 + P_{lsplit} \cdot 4$. Now when the new separator for the split is inserted in the parent index node ($CPU_{ninsert}$), this will result in a split with probability P_{nsplit}. We assume that sibling pointers exists at the index level as well, and ignore the unlikely event of higher level node splits. The final values for a B^+-tree are given in Fig. 8.

$$DA_{binsert} = DA_{random} + 1 + P_{lsplit}\ (\ 4 + P_{nsplit} \cdot 4\ )$$

$$\begin{aligned} CPU_{binsert} = {} & CPU_{random} + CPU_{linsert} + CPU_{pgovhd} + \\ & P_{lsplit}\ (\ CPU_{lsplit} + CPU_{ninsert} + 4 \cdot CPU_{pgovhd}) + \\ & P_{lsplit} \cdot P_{nsplit}\ (\ CPU_{nsplit} + CPU_{ninsert} + 4 \cdot CPU_{pgovhd}) \end{aligned}$$

Fig. 8. Disk access (DA) and CPU costs for Insert in a B^+-tree

Now consider the case of an insert to an *SB*-tree. When a leaf split occurs, we may find no page on the containing multi-page block to take the new sibling leaf; thus, with probability P_{bsplit} we will have to read in the entire multi-page block ($DA_{\text{long}(L)}$), allocate a new multi-page block, move $L/2+1$ nodes to it and write it out ($DA_{\text{long}(L/2)}$), then move an average of $L/4$ blocks into lower positions on the original block ($((3L/4)+1)\cdot CPU_{\text{move}}$) and write it out ($DA_{\text{long}(L/2)}$). To update each of the offspring pointer entries in the parent node takes CPU_{pchange} so we allow $((3L/4)+1)\cdot CPU_{\text{pchange}}$) for this chore. We note that the contribution to the cost of insert as a result of a leaf level block-split will occur only with probability $P_{\text{lsplit}}\cdot P_{\text{bsplit}}$. We ignore the contribution associated with a higher level node block-split, since the multiplier in that case would be $P_{\text{lsplit}}\cdot P_{\text{nsplit}}\cdot P_{\text{bsplit}}$. As we will see, the probability P_{nsplit} is extremely small in most real situations.

$$DA_{\text{sbinsert}} = DA_{\text{random}} + 1 + P_{\text{lsplit}}(4 + P_{\text{nsplit}}\cdot 4 + P_{\text{bsplit}}(DA_{\text{long(L)}} + 2DA_{\text{long(L/2)}}))$$

$$CPU_{\text{sbinsert}} = CPU_{\text{random}} + CPU_{\text{linsert}} + CPU_{\text{pgovhd}} + P_{\text{lsplit}}\,(CPU_{\text{lsplit}} + CPU_{\text{ninsert}} + 4\cdot CPU_{\text{pgovhd}}) + P_{\text{lsplit}}\cdot P_{\text{nsplit}}\,(CPU_{\text{nsplit}} + CPU_{\text{ninsert}} + 4\cdot CPU_{\text{pgovhd}}) + P_{\text{lsplit}}\cdot P_{\text{bsplit}}\,((3L/4)+1)\,(CPU_{\text{move}} + CPU_{\text{pchange}})\,)$$

Fig. 9. Disk access (DA) and CPU costs for Insert in an *SB*-tree

In Theorem 2 of [17], an upper bound is derived for probability of a split, borrow or merge operation (called "unsafe" operations) occurring as a result of an arbitrary insert or delete of an entry (it is assumed that insert and delete operations are equally likely). The formula is given in terms of the parameters b, f and b, defined in our Definition 2.1. The Theorem concludes that when $f=2/3$, we can bound the probability of any unsafe operation by:

$$P_{\text{unsafe}} \leqq 2/((2/3)\,\beta+1)(b+1).$$

Taking the typical value of $b=20$ at the leaf level (for record entries of 200 bytes), we may choose β to have the value 6, and determine that $P_{\text{lsplit}} \leqq 2/105$. With a typical value of $b=340$ at the index node level, we may choose β to have the value 12, with the result that $P_{\text{nsplit}} \leqq 2/3069$. A multi-page block has a typical value of $b'=64$, and we may choose a rather large value of β', say 18, to give a low probability of block-split, $P_{\text{bsplit}} \leqq 2/845$.

Using these values for b and β for formulas which occur in Figs. 8 and 9, assuming $L=64$ and DA_{random} values for 1 million records, we can calculate

the contribution of the additional terms associated with inserts into the *SB*-tree. In the estimate of disk accesses for a B^+-tree insert, we see that:

$$\mathrm{DA}_{\mathrm{binsert}} = \mathrm{DA}_{\mathrm{random}} + 1 + P_{\mathrm{lsplit}}(4 + P_{\mathrm{nsplit}} \cdot 4)$$

has an approximate value of: $1+1+(2/105)\ (4+8/3069)=2.0762$, whereas the formula for the *SB*-tree adds a new term of the form:

$$P_{\mathrm{lsplit}} \cdot P_{\mathrm{bsplit}}(\mathrm{DA}_{\mathrm{long}(L)} + 2\,\mathrm{DA}_{\mathrm{long}(L/2)})$$

which can be estimated as approximately: $(2/105)(2/845)(5+8)=0.00059$, an insignificant addition. In the case of CPU cost, if we compare the formula for B^+-tree insert.

$$\begin{aligned}\mathrm{CPU}_{\mathrm{binsert}} = {} & \mathrm{CPU}_{\mathrm{random}} + \mathrm{CPU}_{\mathrm{linsert}} + \mathrm{CPU}_{\mathrm{pgovhd}} \\ & + P_{\mathrm{lsplit}}(\mathrm{CPU}_{\mathrm{lsplit}} + \mathrm{CPU}_{\mathrm{ninsert}} + 4 \cdot \mathrm{CPU}_{\mathrm{pgovhd}}) \\ & + P_{\mathrm{lsplit}} \cdot P_{\mathrm{nsplit}}(\mathrm{CPU}_{\mathrm{nsplit}} + \mathrm{CPU}_{\mathrm{ninsert}} + 4 \cdot \mathrm{CPU}_{\mathrm{pgovhd}})\end{aligned}$$

with the additional term associated with the *SB*-tree,

$$\mathrm{CPU}_{\mathrm{bsplit}} = P_{\mathrm{lsplit}} \cdot P_{\mathrm{bsplit}}((3L/4)+1)(\mathrm{CPU}_{\mathrm{move}} + \mathrm{CPU}_{\mathrm{pchange}})$$

we are at first impressed with the large multiple for the actions which must be performed, $((3L/4)+1)$. However, if we compare this term to the single term associated with leaf split, $P_{\mathrm{lsplit}}(\mathrm{CPU}_{\mathrm{lsplit}} + \mathrm{CPU}_{\mathrm{ninsert}})$, and observe that the CPU cost for $\mathrm{CPU}_{\mathrm{move}}$ is almost certainly less than the cost $\mathrm{CPU}_{\mathrm{split}}$ and $\mathrm{CPU}_{\mathrm{pchange}}$ is similarly less than $\mathrm{CPU}_{\mathrm{ninsert}}$, we note that the ratio $\mathrm{CPU}_{\mathrm{bsplit}}/\mathrm{CPU}_{\mathrm{lsplit}}$ is no more than

$$P_{\mathrm{bsplit}} \cdot ((3L/4)+1) = (2/845)(48+1) = 0.1160.$$

Thus, the CPU time for the block split adds a relatively small effect to the cost of leaf splitting, itself only a fraction of the total cost of insert.

3.1.5 Deletion. Deletion of an entry in a B^+-tree is quite comparable to an insert in performance. It entails searching from the root for the proper leaf location, removing the desired entry from the leaf structure, and writing back the leaf node modified. With probability P_{llow}, we will find the leaf has fallen to a stage below the lower allowable limit, *a*, and we will perform either a borrow or a merge operation (we say that the probability of a borrow being successful is P_{lborrow}, and thus the probability that a merge is required is $(1\text{-}P_{\mathrm{lborrow}})$). In the case of a borrow, we will copy some entries of the neighboring node into the leaf from which the delete occurs; this changes the separator above ($\mathrm{CPU}_{\mathrm{schange}}$) and then both nodes and the parent must be written out. If a merge is performed, an additinal sibling node must be read to update the sibling pointer. Now when the separator between the merged nodes is deleted from the parent index node ($\mathrm{CPU}_{\mathrm{ndelete}}$), this will result in a borrow with probability P_{nlow}, the borrow is successful with probability P_{nborrow}, and so on. We assume that sibling pointers exist at the index level as well, and ignore the

$$DA_{bdelete} = DA_{random} + 1 + P_{llow}\,(\,P_{lborrow}\cdot 2 + (1-P_{lborrow})\cdot 3\,) + P_{llow}\cdot(1-P_{lborrow})\cdot P_{nlow}\,(\,P_{nborrow}\cdot 2 + (1-P_{nborrow})\cdot 3)$$

$$\begin{aligned} CPU_{bdelete} = {} & CPU_{random} + CPU_{linsert} + \\ & P_{llow}\,(\,P_{lborrow}\cdot(CPU_{lborrow} + CPU_{schange} + 2\cdot CPU_{pgovhd}) + \\ & \quad (1-P_{lborrow})\cdot(CPU_{lmerge} + CPU_{ndelete} + 3\cdot CPU_{pgovhd})) + \\ & P_{llow}\cdot(1-P_{lborrow})\cdot P_{nlow}(\,P_{nborrow}\cdot(CPU_{nborrow} + \\ & \quad CPU_{schange} + 2\cdot CPU_{pgovhd}) \\ & \quad + (1-P_{nborrow})\cdot(CPU_{nmerge} + CPU_{ndelete} + 3\cdot CPU_{pgovhd})) \end{aligned}$$

Fig. 10. Disk access (DA) and CPU costs for Delete in a B^+-tree

$$DA_{sblow} = P_{llow}\cdot(1-P_{lborrow})\cdot P_{blow}\cdot P_{bborrow}(N_{borrow}) + P_{llow}\cdot(1-P_{lborrow})\cdot P_{blow}\cdot(1-P_{bborrow})(N_{bmerge})$$

$$\begin{aligned} CPU_{sblow} = {} & P_{llow}\cdot(1-P_{lborrow})\cdot P_{blow}\cdot(\\ & P_{bborrow}(N_{borrow}\cdot(CPU_{move}+CPU_{pchange}+CPU_{pgovhd})) + \\ & (1-P_{bborrow})(N_{bmerge}\cdot(CPU_{move}+CPU_{pchange}+CPU_{pgovhd})) \end{aligned}$$

Fig. 11. Disk access (DA) and CPU added term for block low in delete

unlikely event of higher level node operations. The final values for a B^+-tree are given in Fig. 10.

In the case of a delete from an *SB*-tree, we must add to the DA and CPU formulas of Fig. 10, additional terms representing the possibility that a block falls below its low point, and a borrow or merge must occur. These new terms are given in Fig. 11.

As in the case of insert, there are relatively large numbers, N_{borrow} and N_{bmerge}, which represent the number of nodes moved in a block, either in a borrow or in a merge. These are certainly bounded above by $L/2$, and with the assumptions of our analysis of insert, we see that

$$P_{llow} \leqq 2/105, \quad P_{nlow} \leqq 2/3069, \quad \text{and} \quad P_{bsplit} \leqq 2/845.$$

Assuming nominal values $P_{lborrow} = P_{nborrow} = 1/2$ (the results are relatively insensitive to these values between 0 and 1), we have:

$$DA_{bdelete} \approx 1 + 1 + (2/105)(5/2) + (2/105)\cdot(1/2)\cdot(2/3069)\cdot(5/2) = 2.0476$$

whereas the formula for the *SB*-tree adds a new term with value bounded by:

$$DA_{sblow} \approx (2/105) \cdot (1/2) \cdot (2/845) \cdot (32) = 0.0007$$

an insignificant addition. In the case of CPU cost, we can easily estimate an upper bound on the ratio $CPU_{sblow}/CPU_{bdelete}$, copying the method use in insert.

$$CPU_{sblow}/CPU_{bdelete} \leqq P_{blow} \cdot (L/2) = (2/845)(32) = 0.0757.$$

Thus, the CPU time for handling the block low condition adds a relatively insignificant effect to the cost of handling the leaf low condition, itself only a fraction of the total cost of delete.

3.1.6 Summary: comparing the SB-tree with the B^+-tree. The previous sections have gone to show that the *SB*-tree handles long range retrievals more efficiently than the B^+-tree, with a factor of about a 10 to 1 improvement in both elapsed time for disk access and CPU usage. In other operations, the *SB*-tree matches *B*-tree performance in random retrievals and entails no more than an average 5% extra cost in insert and delete operations, as a result of required block balancing.

The conclusion we draw is that in systems where long range retrievals take up significant system resources, an *SB*-tree will result in important resource savings. We next compare performance of the *SB*-tree with that of the bounded disorder file (BD file), introduced in [11] by Lomet.

3.2. Comparison with bounded disorder file performance

The BD file is a storage structure in which an index search in a B^+-tree leads to a large data node of multiple disk pages. The data node is partitioned into a number of distinct parts, called buckets, of either two or three pages each. After the index search has determined the correct node, hashing is used to select the bucket of the data node in which the record is to be stored – thus only the bucket needs to be accessed from disk, rather than the entire node. In hashing to the bucket, sequentiality is sacrificed within the node but is preserved between nodes (thus bounding the "disorder"). The point of the BD file structure is to guarantee an upper bound on effective depth of 1 (plus a small fraction for occasional access to an "overflow bucket") by choosing the data nodes large enough so that the B^+-tree directory level is bounded in size and can always be kept completely in memory. As a result, BD file random retrieval performance is a significant improvement on that of a B^+-tree in very large files where the B^+-tree effective depth is much greater than 1. (See Fig. 3 for a graph of D_E vs. the number of bytes at the leaf level. We note that the number of page buffers *B* assumed in this graph differs from that assumed in [11]; this will be discussed further in Sect. 4.)

To accommodate new insertions in a BD file, the buckets of a data node form an extendible hashing scheme. All but one of the buckets are reached directly by hashing on the key sought, but one bucket is designated as an "overflow bucket". When an insert is performed which hashes to a full bucket in the node, the overflow bucket is used to hold the new record. When the overflow bucket fills up, this triggers an expansion step, with two different phases which together double the size: a node with *n* 2-page buckets is expanded into a node with *n* 3-page buckets, while a node with *n* 3-page buckets is expanded into two nodes with *n* 2-page buckets. The resulting utilization (fullness) of

the BD file space is about 0.70: A somewhat more frequent reorganization than takes place in B^+-trees (twice for each doubling in size) makes up for the lower space utilization associated with hashing. However, as we see in Fig. 12 below, the effect of more frequent reorganization on increasing the number of disk accesses during insert is quite small.

Lomet differentiates carefully between two different forms of range retrieval: "range search", where the records must be retrieved one at a time in the range (k_s, k_e), but the order of retrieval is not important, and "key sequential access" where the records must actually be delivered in sort order by key. The common uses of merge join and ordered user display seem to require the more difficult key sequential access, although Lomet points out that for many uses. "By changing algorithms, it is frequently possible to use the results of a range search, without the extra cost of putting the records in key order." To support key sequential access, BD file records within each bucket are maintained in key order. Since records are ordered from one node to another, a total ordering can be achieved by performing a "merge" on the ordered sequences from the different buckets of a node, as in the final stage of a multi-merge sort. This is best effected by using a "heap" structure on the smallest keys in each bucket, successively removing the minimum, inserting the next in that bucket, and reorganizing the heap; each such heap reorganization takes $\log_2 k$ steps, where k is the number of buckets. Note that in other sections of this paper, the term "range retrieval" is used generically to denote either "range search" or "key sequential access", since these distinctions make no difference with *B*-tree and *SB*-tree structures.

Leaf Level Size in bytes	BD file Search	SB-tree Search	BD file Insert	SB-tree Insert
10M	0.70	0.69	1.80	1.79
80M	0.97	0.96	2.08	2.06
640M	1.01	1.59	2.11	2.69
2.56G	1.01	1.88	2.11	2.98
10.24G	1.01	1.97	2.11	3.17

Fig. 12. Comparison of disk accesses for BD file and *SB*-tree

The BD file exhibits superior performance characteristics in a number of areas. In Fig. 12 we compare the disk access behavior for a BD file with 32 buckets (thus either 64 or 96 pages per node) with that of an *SB*-tree with 64 page nodes per block. We assume the BD file can take advantage of extra buffer space for small files, like the *SB*-tree.

For very large files, where the effective depth of an *SB*-tree becomes significantly greater than one, the guarantee to keep all of the directory level of the BD tree in memory is an important advantage. The obvious conclusion of BD tree superiority is perhaps weakened by the consideration that the cost for a memory resident directory should enter into the calculation, as dictated by the Five Minute Rule [9]. Presumably the *SB*-tree could also increase its multi-

page block size and require its directory level to be memory resident, rather than depending on an LRU buffer scheme which causes an effective depth greater than one. Delete performance figures for the two structures are not included in the table of Fig. 8 because a delete algorithm for the BD file is not presented in [11]. Lomet certainly has an algorithm in mind, and it is inappropriate to enter here into an analysis of this unpublished method; however, the author believes the *SB*-tree approach to have a performance advantage.

A major strength of the BD file structure is in the area of long range retrievals. Like the *SB*-tree, BD file nodes are designed to be read in a single long disk access, so for very long ranges the disk access time is comparable for the two structures. It is in the area of shorter range retrievals where the BD file structure shows some weak performance characteristics in terms of disk access.

If we assume that the m records in the key range (k_s, k_e) lie on a small number of leaf nodes of an *SB*-tree, say one to four pages where the cost of reading in each of the pages individually is superior to a multi-page block read, the *SB*-tree will support this alternate approach. The BD file, however, must read in all the buckets in the multi-page node in order to provide the records in the key range requested; this effort is necessary whether the records are to be provided for a range search or in key sequential order. To perform the requested retrieval, it is necessary to perform the bucket merge mentioned earlier for all buckets in the node at least through the end of the range desired. Thus a multi-page node read is always needed.

In addition to requiring more time, this approach degrades memory buffer utilization in circumstances where short range retrievals of this kind are common, since in the example given earlier either 64 or 96 pages must be read in rather than the two or three pages required by the *SB*-tree.

The CPU cost to merge the multiple buckets of the node represents another significant inefficiency for either short or long range retrieval in the BD file. In a short range retrieval where the entire range to be retrieved falls in a single node, the midpoint of the range retrieved will, on the average, fall at the midpoint of the total range for the node in which it sits; a merge must be performed from the beginning of the node to the end of the range retrieved, and thus the number of entries to be merged is about half the entries in the node plus half the entries in the range, $m/2$. We can estimate the number of entries in the node as the same number we would have in an *SB*-tree, $F_{\text{bleaf}} \cdot L \cdot F_{\text{leaf}} \cdot P/R$, assuming a comparable number of pages between the *SB*-tree block and the node of the BD file, with comparable utilization. The CPU cost as each entry is generated is what is needed to remove the minimum of a heap, advance the relevant bucket cursor, and insert the next entry under the cursor back into the heap. We represent this as CPU_{heap}, a cost which is logarithmic in the number of buckets in the node. When the entry generated is one of the entries in the desired range, additional retrieval represented by CPU_{next} is entailed. The total CPU cost for such a short range retrieval in a BD file is given in Fig. 13.

$$\text{CPU}_{\text{BDrange}} = \text{CPU}_{\text{BDsearch}} + m \cdot \text{CPU}_{\text{next}} + \text{CPU}_{\text{lgovhd}} + (F_{\text{bleaf}} \cdot L \cdot F_{\text{leaf}} \cdot P/(2R) + m/2) \cdot \text{CPU}_{\text{heap}}$$

Fig. 13. BD file CPU cost for short range retrieval (lower bound)

We are ignoring here the possibility that a small range retrieval might fall across two nodes of the BD file, which would nearly double all costs. Now if we compare this to the CPU cost for an *SB*-tree for such a range (equivalent to that of a B^+-tree, in Fig. 4), we have:

$$\text{CPU}_{\text{SBrange}} = \text{CPU}_{\text{random}} + m \cdot \text{CPU}_{\text{next}} + (\text{DA}_{\text{brange}-1}) \cdot \text{CPU}_{\text{pgovhd}} .$$

For longer range retrievals, the $\text{CPU}_{\text{pgovhd}}$ term in the *SB*-tree retrieval is replaced by a term for $\text{CPU}_{\text{lgovhd}}$. Now it is clear that $\text{CPU}_{\text{BDsearch}}$ is comparable with $\text{CPU}_{\text{random}}$ (we assume binary node search); the total overhead for page reads is usually somewhat greater in the *SB*-tree but this should not be significant. The additional term for merging entries in Fig. 3 would seem to result in significantly more CPU use, however. For short ranges in the BD file we need to merge entries from 32+ pages, where the *SB*-tree range retrieval only needs to access one or two pages of entries. In Fig. 14 we present a very approximate table of CPU usage in instructions (ins) vs range size m for the two structures. We assume 1 million 200 byte leaf entries and that $\text{CPU}_{\text{search}}$ requires 300 ins for setup and about 25 binary search steps of 8 ins each, CPU_{next} requires 20 ins (with some possibility of returning information), $\text{CPU}_{\text{pgovhd}}$ requires 2000 ins, $\text{CPU}_{\text{lgovhd}}$ requires 5000 ins, and CPU_{heap} requires 200 ins. Note that most of these estimates (aside from that for *I/O* overhead), appear quite optimistic when compared to actual measurements of the product DB2, for example (see [13]).

Range size m	$\text{CPU}_{\text{BDrange}}$	$\text{CPU}_{\text{SBrange}}$
10	96300	3986
20	97500	5614
40	99900	6300
160	113300	8700
640	171900	18300

Fig. 14. Approximate CPU usage in a range retrieval of m items

3.3 Summary of performance analysis

The preceding discussion has gone to show that the *SB*-tree has significant performance advantages over a B^+-tree in a dynamic file when long range retrievals make up an appreciable fraction of the work mix. The BD file has advantages over both the B^+-tree and the *SB*-tree in random-access search, insert and delete (with search from the root) where very large files are involved, although a change in node size and directory residence by the *SB*-tree could allow it to share this advantage. The BD file shares the disk access advantage of the *SB*-tree over the B^+-tree for extremely long range retrievals. However, the BD file has performance disadvantages in dealing with files which are highly

varying in size (growing and shrinking) and in relatively short range retrievals, as well as CPU disadvantages in all range retrievals. The *SB*-tree would seem to be the structure of choice in cases where range retrieval, both long and relatively short, make up a large proportion of the retrieval effort, and modifications resulting in node splits and merges occur.

4. Other considerations

This section contains short discussions of several points of interest.

4.1 Disk space utilization

It should be pointed out that we are paying a certain amount in terms of disk space utilization for the high speed range-retrieval I/O associated with the *SB*-tree. The reason is that there are two distinct levels on which we do not have dense packing. First, the nodes on disk can only be assumed to be 70% full (and as we explain in the next section, even this is optimistic because of the slack factor β), and second, the blocks on disk can only be assumed to have 70% (or less) of their nodes utilized. This effectively squares the fullness factor, leading to a disk storage utilization which is probably less than 50% in the structure we have been investigating. It is important to realize, however, that although there is a lower utilization of disk media on the lower levels of the *SB*-tree than there is in an equivalent *B*-tree, *the SB-tree is not of greater depth.* The depth of the tree is determined by the fanout of the nodes in use, and therefore the nodes on the multi-page block which are not in use have no effect on the depth (except for the small effect necessary to contain the marker entry M in the directory nodes).

There is another consideration which makes the storage media utilization of somewhat less importance. In many situations of reasonably common disk access (as explained in [9]), we find that the gating cost factor in purchasing disk is in arm movement to provide sufficiently frequent I/O service, rather than in disk storage capacity. It is common in transactional systems, for example, that we must place high access data of only a few hundred megabytes to be serviced by each disk arm to maintain the needed I/O access rate, leaving a gigabyte or more unused. In general we can only use this added storage capacity for extremely low access purposes, such as archival storage. A *B*-tree or *SB*-tree usually has extremely high rates of access in comparison with other forms of data, and it seems that the decreased disk storage utilization is no great hardship under such circumstances. In cases where low I/O utilization is encountered, we are probably dealing with a low-update workload, where a variation such as that covered below in Sect. 4.2.1 would be of value in improving utilization.

4.2 Variations from the SB-tree structure

There are a number of areas where it is appropriate to entertain design variations from the *SB*-tree structure presented here. For example, a number of B^+-trees with variable length key and leaf entries have been implemented, and this feature can be quite valuable, both in terms of space utilization and disk access time, where closer packing results in fewer range retrieval accesses for example. A binary search in a node with variable length entries can still take place, using

a table of fixed length offsets to variable length entries in collation order. The approach in [17] to keeping the number of entries in a range (a, b) with a slack factor β can be modified easily to deal with a range which measures "fullness" in bytes. It would seem that the probabilistic cost analysis can be carried over by making an assumption of an "average" length of the keys and leaf entries.

It is clear that the *SB*-tree depends on the analysis of [17] for many of its efficiencies. In this regard, we should note that the analysis in [1] which derives a utilization factor of about 70% for B^+-trees, applies to the standard B^+-tree, with slack factor $\beta=0$, and our use of this figure has been somewhat inaccurate. Further analysis is needed to determine utilization for non-zero β.

4.2.1 Multi-node SB-tree reorganization. Recall that in Sect. 2.1 we settled on a definition for the *SB*-tree which avoided the approach of the B^+-tree of reorganizing 2 nodes to 3 in a split, and later 3 nodes to 2 in a merge. It was pointed out that the survival interval of a B^+-tree (average number of inserts and deletes between merges and splits) was reduced from that of a B^+-tree – this is because each node now is guaranteed to have its number of entries in a smaller range, with a lower limit of 2/3 full, so there is less "slack". However, we note that by avoiding a fullness lower bound of 2/3, we are cutting back on the number of reorganizations from inserts and deletes at the expense of increasing the number of nodes which must be traversed in a multi-node range retrieval (because the nodes are less full in the more slack case on the average). In a situation where long range retrievals outnumber inserts and deletes by a large factor, we need to reconsider this decision.

It is of particular interest that an *SB*-tree is able to support greater ratios than 2/3 with relatively efficient disk access. We can perform multi-page block I/O to allow reorganization among K nodes (out of L) or to split to $K+1$ or merge to $K-1$, keeping all nodes nearly full. Since the I/O's are relatively efficient, the major aspect mitigating against such an approach is the CPU time needed to perform a large number of entry moves, choose all new separators, and modify them in the parent node(s). This CPU cost is especially significant since as we keep the nodes more and more full we lose any slack factor we had and reorganizations become more frequent.

In spite of this, it would seem that an ideal *SB*-tree implementation would permit a user choice as to the number of nodes in a reorganization, up to a reasonable limit such as the minimum number on a block. The parameter is not particularly hard for the file manager to supply, and in designing the access method we should try not to prejudge the relative frequency of range-retrievals in comparison to updates. Over a long enough period of time even extremely low rates of insert and delete will bring about split and merge reorganization. But with extremely low update rates, the frequency of splits from K to $K+1$ nodes or merges from K to $K-1$ is low, even with large values for K, while the performance to be gained from performing range retrievals from nearly full nodes may be a good deal more significant.

4.3 Buffering

The concept of effective depth of a B^+-tree presented in Sect. 3.1.1 is the first rough treatment known to the author to predict random access disk accesses

for a B^+-tree in a normal buffered environment. We note that the concept of level W is central to the development, where levels below W are so numerous as to be basically 100% out of buffer and levels above are thought of as 100% present in buffer. Several buffering schemes have been suggested which are not LRU, in the sense that nodes on a thread below level W are released relatively quickly, thus leaving more memory buffers for nodes at level W, and other pages which will be more likely to be accessed again before they are dropped from buffer. Unfortunately, although this idea is well covered in theory, the author is unaware of any commercial implementation.

It should also be noted that the BD file leaf node buckets should be released relatively quickly under most circumstances. However, it is probably not appropriate to free buffers of buckets right away, but rather to keep enough of them in memory so that all the buckets of a node can be resident. This is a valuable approach in circumstances where a sequence of keys are presented for random retrieval which in fact are strictly ordered, but where the system has no warning of this. Thus random access will be performed, rather than some attempt at sequential access; however, we can expect continuous buffer "hits" for successive accesses after the first, so long as some reasonable approach is used to keep pages referenced extremely recently in memory buffers.

4.3.1 The five minute rule and usable buffer space. Recall that an assumption used in deriving the results of Fig. 3, the graph of the effective depth of a B^+-tree vs the number of bytes at leaf level, was that $B=1000$, that is that there were 1000 memory buffer pages or 4 M bytes dedicated to B^+-tree node pages. In [11], on page 527, there is an explicit statement that, "... the 4 M bytes needed by a memory resident B-tree index is almost surely not feasible". Some justification for the $B=1000$ number is therefore appropriate.

In [9], a formulation was advanced for the tradeoff between memory costs and the cost of disk access, to derive the frequency with which a memory page should be accessed if it is to remain in memory buffer. The idea is that memory buffers are provided to save accesses to disk; any frequency of disk access, such as one access per second, can be charged a fair "rent" in terms of the resources it requires; at the same time, a megabyte of memory has a cost and therefore a fair "rent" for the duration of its useful life. It would seem that if a disk page of storage is accessed with some sufficient frequency, then it is economical to purchase memory if necessary to ensure that the 4 K bytes involved stay in memory buffers. On the other hand, if it falls below that frequency, we would be willing to throw the page out of buffers and take the cost of reading it back in when it is needed, purchasing more disk access arms if necessary. The "indifference point" in this tradeoff turns out to be about 100 seconds for a 4 K page block. Pages referenced more frequently than that should remain in buffer. The point was also made that the length of time is increasing as memory prices go down faster than disk access costs: the prices used for the paper were from 1986, and an expansion factor of 60 in twenty years was mentioned.

Given a 1000 page buffer, we require page accesses 10 times per second evenly spread over the pages in the buffer in order to "pay the rent" on the buffers. The analysis becomes a bit complicated, but what this roughly means is that in order for 1000 buffers to be economical, we need about 10 random searches per second down through the B^+-tree. This seems like a not unreason-

able access rate for a very large tree containing the records of a relatively important file. As time passes, even less activity will be necessary to justify this buffer size. Note that for many years the economic use of IBM memory was artificially constrained by a 16 megabyte virtual memory limit, and users who do not have XA systems are still constrained in that way. This may serve to explain the statement in [11] to the effect that 4 M bytes for buffers was not feasible.

4.3.2 Memory buffering performance in sequential access. A key consideration in memory buffering is whether we need to keep more than one size of buffer: single page buffers for page-nodes and larger sizes for multi-page blocks. It is a common property of modern computer systems that a multi-page read may be accomplished in a scatter/gather form to several non-contiguous memory locations (as mentioned also, for example, in [11]). However, Dieter Gawlick has pointed out that experiments in ultra high speed striped sequential file processing (with effective transfer rates of over a hundred megabytes per second) found the CPU needed to allocate over 25000 4 K buffers per second an important drain on CPU system resources [7, 4]. At the same time, if buffers capable of accommodating full multi-page blocks are invariably used to accommodate partially full blocks, the value of compression of data onto the early pages of the block will be lost so far as memory utilization is concerned (there will still be value in terms of a reduced channel transfer time). The value of memory space in terms of CPU is another tradeoff investigated in [9]: as a rather rough rule of thumb, with current costs of memory and CPU we should be indifferent to a tradeoff of 1 instruction per second with 10 bytes of memory. However, if CPU time becomes the gating factor in high speed sequential transfer rate, the cost tradeoff may take second place to response time requirements.

To deal with the need for fast buffer allocation without wasting memory space, various alternatives suggest themselves. One possibility is to use variable lengths of long buffers, with sizes ranging, for example, from 32 to 64 pages, a service chain for each, and a first fit in increasing size algorithm if all buffers of the exact size needed are in use. This will certainly make much more efficient use of memory space than a simple 64 page buffer pool, and should be quite efficient in terms of CPU time to locate an appropriate buffer and create the needed channel program.

In the same spirit we may allocate disk space for new storage blocks in multiple sizes, say 32, 48 and 64 page blocks (as was suggested also in [11]). Using multiple block sizes in this way is expensive in terms of more frequent complex rearrangements (moving a block-node from one disk segment to another), and would presumably be appropriate only for indexes with infrequent updates at current low prices for disk media.

5. Conclusions

We have introduced the *SB*-tree, a variant of the *B*-tree, optimized for sequential multi-page disk access during long range retrievals. In comparison to the *B*-tree, the *SB*-tree exhibits essentially no performance degradation for single-page access random key retrieval, and a degradation of at most 5% for insert and delete operations. Performance comparison with the BD file reveals that the *SB*-tree has advantages in situations where relatively short range retrievals up

to several pages make up a large proportion of the retrieval effort, or where modifications resulting in node splits and merges occur with some frequency. The sequentially optimized performance characteristics of the *SB*-tree are designed to be resilient to node splits and merges, and the *SB*-tree is therefore appropriate for situations where frequent *B*-tree reorganizations are not possible.

The performance analysis of random key retrieval led us to develop the concept of "Effective Depth" of a *B*-tree or *SB*-tree. This measure should be useful in a number of other situations where performance analysis is undertaken for disk resident tree structures.

Acknowledgements. The author would like to acknowledge the help of several colleagues who offered valuable advice during the preparation of this paper. These are: Mei Hsu, Toby Lehman, Elizabeth O'Neil, and particularly Dieter Gawlick and Dave Lomet, who made many substantial suggestions of value. The author would also like to thank the referees for a number of useful points.

References

1. Baeza-Yates, R.A.: Expected behavior of B^+-trees under random insertion. Acta Inf. **26**, 439–471 (1989)
2. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf. **1**, 173–189 (1972)
3. Bayer, R., Unterauer, K.: Prefix *B*-trees. ACM Trans. Database Syst. **2 (1)**, 11–26 (1977)
4. Chen, P.M., Gibson, G.A., Katz, R.A., Patterson, D.A.: An evaluation of redundant arrays of disks using an Amdahl 5890. Proceedings of the 1990 SIGMETRICS Conference on Measuring and Modeling of Computer Systems, May 1989
5. Chen, P.M., Patterson, D.A.: Maximizing performance in a striped disk array. 17th International Symposium on Computer Architecture (SIGARCH 1990)
6. Comer, D.: The ubiquitous *B*-tree. Comput. Surv. **11**, 121–137 (1979)
7. Gawlick, D.: DEC, private communication (March 1989)
8. Gray, J.: "DISCS", Notes for talk Sponsored by DEC Cambridge Research Lab and Greater Boston SIGMOD, Cambridge, MA, February 6, 1989
9. Gray, J., Putzolu, F.: The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. Proceedings of the 1987 ACM SIGMOD Conference, pp. 395–398
10. INGRES SQL Reference Manual. Relational Technology, Alameda, CA
11. Lomet, D.B.: A simple bounded disorder file organization with good performance. ACM Trans. Database Syst. **13 (4)**, 525–551 (1988)
12. O'Neil, P.E.: Revisiting DBMS benchmarks. Datamation September 15, 47–53 (1990)
13. O'Neil, P.E.: The set query benchmark. The performance handbook for database and transaction processing systems. Hove: Morgan Kaufmann 1991
14. Teng, J.Z., Gumaer, R.A.: Managing IBM database 2 buffers to maximize performance. IBM Syst. J. **23 (2)**, 211–218 (1984)
15. Wiederhold, G.: Database design. London: McGraw Hill 1983
16. Wodnicki, J.M., Kurtz, S.C.: GPD performance evaluation lab database 2 version 2 utility analysis. IBM Document Number GG09-1031-0, September 28, 1989
17. Zhang, B., Hsu, M.: Unsafe operations in *B*-trees. Acta Inf. **26**, 421–438 (1989)