Space-Efficient Bounded-Entry B+-Tree

[Extended Abstract]

Christoph Schwering
Hainbuchenstr. 6
52074 Aachen, Germany
schwering@gmail.com

ABSTRACT

A B-tree is an order-preserving index structure for block oriented storage devices, because each node of the tree can reside on a single block. The B-tree was first described by Bayer and McCreight [1]. Several variants of the B-tree appeared in the aftermath, in particular the B^+-tree [2, 3], which differs from the B-tree in that it carries the values only in its leaves.

This paper proposes a B^+-tree (which will be called B-tree for short in the following) that is capable of storing bounded-length key/value-entries without wasting space and without the need of overflow spaces. This is achieved by restricting the entries to a maximum length of $\frac{1}{4}$ of a block's size and posting the invariant that each node except for the root is filled by a degree of at least $\frac{3}{8}$ with entries.

This idea has been mentioned shortly by Knuth [3], but it seems to the me that concrete length-bounds and node-invariants have not been published. The reason for this probably is that it is trivial for smart people, but sadly it's not for me.

1. DEFINITION

Definition 1. A *node* N is a set $\{e_1, \dots, e_d\}$ of entries. The degree of N is the count of entries $deg(N) = d$.

Besides the entries, there is some meta data $meta(N)$ that contains the degree $deg(N)$, the neighbors' addresses $left(N)$ and $right(N)$ and its parent's address $parent(N)$. If such a neighbor or parent nodes does not exist, the respective value is $\perp$.

The form of an entry depends on the position of the node. An entry e_i of a *inner node* consists of

- a key value $key(e_i)$,
- an address $child(e_i)$.

An entry e_i of a *leaf node* consists of

- a key value $key(e_i)$, and
- a data value $value(e_i)$.

For inner nodes, we write $e = (k, a)$ and for leaf nodes $e = (k, v)$. If the context permits it, we do not distinguish between addresses and nodes, i.e. $parent(child(e_i)) = N$, for example.

Definition 2. A *B-tree* is defined over a domain K of keys that is fully ordered by $\leq \subseteq K \times K$ and a domain V of values. We write $e_i \leq e_j$ in the following iff $key(e_i) \leq key(e_j)$. The B-tree supports the operations *insertion*, *deletion* and *search*. We can define these operations by looking at the states of the tree:

- In state S_0, the B-tree is empty and for all k, $search(S_0, k) = \perp$ (i.e. no value v for k can be found).
- In a state S_i, the operation $insert(S_i, k, v)$ reaches a state S_{i+1} in which $search(S_{i+1}, k) = v$, if $search(S_i, k) = \perp$. Otherwise, the insertion fails and the B-tree state does not change.
- In a state S_i, the operation $delete(S_i, k)$ returns v and reaches a state S_{i+1} in which $search(S_{i+1}, k) = \perp$.

In all states, the key condition which makes searching efficient must hold for all nodes N in the B-tree:

$$\forall e \in N : child(e) = N' \\ \Rightarrow \forall e' \in N' : e' \leq e.$$

Size Limits

Normal B-trees require their nodes to have a certain fixed minimum and maximum degree, because each node is intended to be stored in one hard disk block of size B (typically 4096 bytes). In contrast, our B-tree requires each node's storage space to be used to a minimum. This enables the B-tree to support *keys and values of variable length*.

Note that for all nodes N, $size_{\text{meta}} = size(meta(N))$ is constant, and for all child addresses a, $size_{\text{addr}} = size(a)$ is constant.

We write $B_{\text{eff}} = B - size_{\text{meta}}$ for the effective node size and leave out the size of meta data in $size(N) = \sum_{e \in N} size(e)$ because we always compare it to B_{eff}. For each $e \in N$, the size is determined by

$$size(e) = \begin{cases} size(k) + size_{\text{addr}} & \text{if } e = (k, a) \\ size(k) + size(v) & \text{if } e = (k, v) \end{cases}.$$

Finally, we require for each node N to hold the requirements depicted in figure 2.

VLDB '10, September 13-17, 2010, Singapore

$$size(e) \le \lfloor \tfrac{1}{4} B_{\text{eff}} \rfloor \quad \text{for all } e \in N \tag{1}$$
$$\lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor \le size(N) \le B_{\text{eff}} \quad \text{for all } N \text{ except the root} \tag{2}$$
$$0 \le size(R) \le B_{\text{eff}} \quad \text{for the root node } R \tag{3}$$

Figure 1: Node filling requirements

Definition 3. A node is *valid* iff it meets the requirements of figure 2.

Remark 1. Note that req. 1 refers to a node element e which consists of either $e = (k, a)$ or $e = (k, v)$. It follows that the *maximum key size* is

$$size(k) + size_{\text{addr}} \le \lfloor \tfrac{1}{4} B_{\text{eff}} \rfloor \quad \text{and}$$
$$size(k) + size(v) \le \lfloor \tfrac{1}{4} B_{\text{eff}} \rfloor.$$

Hence, if the size $size(v)$ of all data values v is bounded by $size_{\text{value}}$, the maximum key size is bounded as follows:

$$size(k) \le \lfloor \tfrac{1}{4} B_{\text{eff}} \rfloor - \max\{size_{\text{addr}}, size_{\text{value}}\}.$$

Remark 2. The B-tree cannot perform worse than a binary B-tree, because even in the worst case, each node except for the root node must hold at least two elements. This is because the minimal node usage is greater than the maximum entry size.

2. THEOREMS

Let there be a B-tree whose nodes are all valid. Assume that due to insertion or deletion of an entry, one of the nodes has become invalid because it violates req. 1. The following theorems show that in all cases the B-tree can be transformed so that all its nodes are valid again.

Node Overflow

THEOREM 1. *A node with entry set $N \cup \{e\}$ with*

$$size(N) \le B_{\mathit{eff}} \text{ but } size(N \cup \{e\}) > B_{\mathit{eff}}$$

can be split into two nodes such that both fulfill req. 2.

PROOF. For $N \cup \{e\} = \{e_1, \ldots, e_n\}$, set $L := \{e_1, \ldots, e_i\}$ such that

$$size(L) \ge \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor \text{ but } size(L \setminus \{e_i\}) < \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$

and $R := (N \cup \{e\}) \setminus L$. Then the sizes are bounded as follows:

$$size(L) \overset{\text{def}}{\ge} \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(L) < \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor + \underbrace{size(e_i)}_{\le \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} \le \lfloor \tfrac{5}{8} B_{\text{eff}} \rfloor$$

$$size(R) = \overbrace{size(N \cup \{e\})}^{> B_{\text{eff}}} - \overbrace{size(L)}^{< \lfloor \frac{5}{8} B_{\text{eff}} \rfloor} > \lceil \tfrac{3}{8} B_{\text{eff}} \rceil$$
$$size(R) = \underbrace{size(N \cup \{e\})}_{\le B_{\text{eff}} + \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} - \underbrace{size(L)}_{\ge \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} \le \lceil \tfrac{7}{8} B_{\text{eff}} \rceil$$

By this, it directly follows that $\lfloor \frac{3}{8} B_{\text{eff}} \rfloor \le size(L), size(R) \le B_{\text{eff}}$, i.e. L and R fulfill req. 2. □

Node Underflow

THEOREM 2. *When an entry e is removed from a non-root-node with entry set L such that $size(L) \ge \lfloor \frac{3}{8} B_{\mathit{eff}} \rfloor$ but $size(L \setminus \{e\}) < \lfloor \frac{3}{8} B_{\mathit{eff}} \rfloor$, either entries from a neighbor can be moved or L can be merged with a neighbor.*

PROOF. Let

$$\delta := \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor - size(L \setminus \{e\})$$

be the space in $L \setminus \{e\}$ that needs to be filled so that $L \setminus \{e\}$ fulfills req. 2. Without loss of generality, we assume that the right neighbor with entry set R of L exists; otherwise, a left neighbor exists and instead of the neighbor's minimal elements its maximal elements must be moved to L.

Depending on $size(R)$, either entries must be moved or nodes must be merged:

1. If $size(R) \le \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta$, the nodes can be merged to a single node $N := (L \setminus \{e\}) \cup R$:

$$size(N) = \overbrace{size(L \setminus \{e\})}^{\ge \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} + \overbrace{size(R)}^{\ge \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} \ge \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(N) = \underbrace{size(L \setminus \{e\})}_{\le \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \delta} + \underbrace{size(R)}_{\le \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta} \le B_{\text{eff}}$$

2. If $size(R) > \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta$, a set $S \subseteq R$ must be moved from R to L. We then set $L' := (L \setminus \{e\}) \cup S$ and $R' := R \setminus S$. The set S must be minimal, i.e.

$$size(L') \ge \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor \text{ but } size(L' \setminus \max S) < \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor.$$

It follows that

$$\delta \le size(S) < \delta + \lfloor \tfrac{1}{4} B_{\text{eff}} \rfloor.$$

Hence, L' and R' meet req. 2:

$$size(L') \overset{\text{def}}{\ge} \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(L') = \underbrace{size(L \setminus \{e\})}_{= \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \delta} + \underbrace{size(S)}_{< \delta + \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} < \lfloor \tfrac{5}{8} B_{\text{eff}} \rfloor$$

$$size(R') = \overbrace{size(R)}^{> \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta} - \overbrace{size(S)}^{< \delta + \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} > \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(R') < size(R) \le B_{\text{eff}}$$

□

3. ALGORITHMS

Procedure 1. To *insert* a key-value-pair (k, v), in the first step the corresponding leaf node is searched (proc 3). Then the node increase handling (proc. 4) is performed.

Procedure 2. To *delete* a key k, in the first step the corresponding leaf node is searched (proc 3). If such a leaf is found, the key-value-pair is removed and then the node decrease handling (proc. 5) is performed.

Procedure 3. The *search* for a key k can be carried out by performing the following steps:

1. Set N to the root node with $N = \{e_1, \ldots, e_d\}$.
2. Choose $n := \arg\max_i k \leq key(e_i)$.
3. If no such n exists
   - and the best future position for k is searched, set $n := deg(N)$ if N is an inner node and set $n := deg(N) + 1$ if N is a leaf,
   - and the value for k is searched, exit with an error.
4. If N is a leaf, return $value(e_n)$.
5. Set $N := child(e_n)$ and go to step 2.

Procedure 4. To *handle the increase of a node* N, the following options are tried; each one is only realized if the resulting nodes are valid.

1. Maybe the node is valid.
2. Try to shift the $\min N$ to $left(N)$.
3. Try to shift the $\max N$ to $right(N)$.
4. Split N into two.

This order chosen to minimize the creation of new nodes (in the first place and disk accesses in the second place).

These steps are in detail (the synchronization and redistributions are described below):

ad 1. Write N to its (old) place and synchronize with parent (proc. 8).

ad 2. Redistribute $left(N)$ and N (proc. 7, redistribution cares about synchronization).

ad 3. Redistribute N and $right(N)$ (proc. 7, redistribution cares about synchronization).

ad 4. Split N to two nodes L and R (proc. 6). R replaces N, while L is a new node. If N was root, create a new root node with the entries L and R and set their parents appropriately (this is the only moment when the *B-tree grows in height*). Otherwise, insert L into its parent and synchronize R with its parent (proc. 1 and proc. 8; insertion of L cares about its synchronization, note that the insertion of L into $parent(L)$ *might have changed* $parent(R)$).

Procedure 5. To *handle the decrease of a node* N, the following options are tried; each one is only realized if the resulting nodes are valid.

1. Maybe N is root and $deg(N) \leq 1$.
2. Try to merge N with $left(N)$.
3. Try to merge N with $right(N)$.
4. Maybe the node is valid.
5. Try to shift $\min right(N)$ to N.
6. Shift $\max left(N)$ to N.

The order is chosen to maximize the removal of nodes (in the first place and disk accesses in the second place).

These steps are in detail (the synchronization and redistributions are described below):

ad 1: If $deg(N) = 0$, do nothing (N must also be a leaf in this case). If $deg(N) = 1$, remove N and make its single child new root unless the root is already a leaf (this is the only moment the *B-tree decreases in height*).

ad 2+3: Let $L = left(N)$ and $R = N$ respectively $L = N$ and $R = right(N)$. Add the the entries of L to R. Note that, in general, $parent(L) \neq parent(R)$. Synchronize R with $parent(R)$ and delete L from $parent(L)$ and remove the node L (proc. 8 and proc. 2; deletion of L cares about its synchronization, note that the synchronization of R with $parent(R)$ *might have changed* $parent(L)$).

ad 4: Write N to its (old) place and synchronize with parent (proc. 8).

ad 5: Redistribute N and $right(N)$ (proc. 7, redistribution cares about synchronization).

ad 6: Redistribute $left(N)$ and N (proc. 7, redistribution cares about synchronization).

Procedure 6. A node $N = \{e_1, \ldots, e_d\}$ is *split optimally* into two nodes L and R at position

$$\arg\max_n \underbrace{\min_n \{size(\{e_1, \ldots, e_{n-1}\}), size(\{e_n, \ldots, e_d\})\}}_{=:d(n)}.$$

The following procedure calculates the optimal split position:

1. Set $n := 0$.
2. If $d(n) > d(n+1)$, exit with n.
3. Otherwise set $n := n + 1$.
4. Go to 2.

Then set $L := \{e_1, \ldots, e_{n-1}\}$ and $R := \{e_n, \ldots, e_d\}$. Set $parent(L) := parent(R) := parent(N)$, $left(L) := left(N)$, $right(L) := R$, $left(R) := L$, $right(R) := right(N)$.

Proof. For the initial setting $n = 1$, L contains no elements. In the n-th iteration, the n-th element is moved from R to L. The procedure stops in the n-th iteration if the $n+1$-th iteration would worsen the situation, i.e. would make the smaller node even smaller. As L grows with each iteration, R would be the smaller node in the $n+1$-th iteration. In any further iteration, R would become even smaller. It follows that no further iteration would improve the situation. □

Procedure 7. To *redistribute two nodes* L and R, they are firstly combined two a single node N (which is overflown). Then N is split optimally to L' and R' (proc. 6). Note that either children move from L to R' or from R to L'. Synchronize L and R with $parent(L)$ and $parent(R)$ (proc. 8; note that the synchronization of L with $parent(L)$ *might have changed* $parent(R)$).

Procedure 8. The following procedure *synchronizes a node N with its parent* $P = parent(N)$. Let $e_i \in P$ with $child(e_i) = N$ be the entry of N in P. The main task of the synchronization is to set $key(e_i) := key(\max N)$. Let e_i' be the entry e_i before its modifications.

The modifications might have the effect that the P is *not valid anymore*:

- If $size(key(\max N)) < size(key(e_i'))$, P might violate req. 2 after the update. A node decrease handling (proc. 5) of P must be performed.
- If $size(key(\max N)) > size(key(e_i'))$, P might violate req. 1 after the update. A node increase handling (proc. 4) of P must be performed.
- Otherwise, P must be written back and, if $i = deg(N)$ and $key(e_i) \neq key(e_i')$, P must be synchronized with its parent (proc. 8).

The increase and decrease handling procedures care about writing back P and further synchronizations.

Note that the increase and decrease handlings might have effect on the neighbors of P, and also on their children and parents. This means that a synchronization of a node L might have the effect that the parent $parent(R)$ of some sibling R of L changes.

4. SUMMARY

Compared to traditional fixed-entry-size B-trees, the presented insertion and deletion procedures are more complex. The reason is that traditional B-trees do not have to shift entries from or to the neighbors when a node has under- or overflown. Even compared to normal B-trees that do this shifting to reduce the count of nodes in the B-tree, the presented algorithms are more costly because shifting entries might also affect the upper level of the B-tree recursively up to the root node.

Besides the high IO costs, this raises concurrency issues because of the higher number of needed locks. However, it should be easy to adopt the concurrency protocol proposed by [4] for traditional B-trees.

The entries can be compressed when req. 2 is relaxed to $deg(N) \geq 2$ for all N except the root.

Obviously, the described B-tree is only suitable for cases in which much more searches than insertion or deletions occur.

5. REFERENCES

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 18: B-Trees, pages 434–454. MIT Press, 2 edition, 2001.

[3] D. E. Knuth. *The Art of Computer Programming*, volume 3: Searching and Sorting, chapter Multiway Trees, pages 481–491. Addison Wesley, 3 edition, 1997.

[4] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.