Dingsbums B⁺-Tree

Christoph Schwering
schwering@gmail.com

October 15, 2008

This article gives a formal description of the B⁺-Tree used in the Dingsbums 6 database system. A B⁺-Tree is an order-preserving index structure for block oriented storage devices. 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. Knuth also mentions a variant that is capable of storing variable-length keys, but does not describe it further [3].

The B⁺-Tree described in this article supports bounded length entries and access by the index number of entries (both features are independent from another, though). Ignoring some rather small internal overhead and the value sizes for the moment, each entry's size is bounded by $\frac{1}{4}$ of the node size in storage elements, and the usage of each node in the tree is at least $\frac{3}{8}$ of its size in storage elements.

1 Definition

Definition 1.1 (Node)**.** 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 $key(e_i)$,
- a count $count(e_i)$, and
- 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 $e = (k, c, a)$ 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. If N is a leaf node, $count(e)$ is defined as 1.

Definition 1.2 (Tree)**.** 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 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 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 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 tree:

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

Furthermore, the counts must be consistent in all states:

$$\forall e \in N : child(e) = N' \\ \Rightarrow count(e) = \sum_{e' \in N'} count(e').$$

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 tree requires each node's storage space to be used to a minimum. This enables the 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 count values c and child addresses a, $size_{\text{count}} = size(c)$ and $size_{\text{addr}} = size(a)$ are 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

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

Finally, we require for each node N two hold the requirements depicted in figure 1.2 on page 2.

Remark 1.3 (Key size). Note that req. 1 refers to a node element e which consists of either $e = (k, c, a)$ or $e = (k, v)$. It follows that the maximum key size is

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

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:

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

Remark 1.4 (Degeneration). The tree cannot perform worse than a binary 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

Theorem 2.1 (Node-Overflow)**.** A node with entry set $N \cup \{e\}$ with

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

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

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

Figure 1: Node filling requirements

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

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

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

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

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

Theorem 2.2 (Node-Underflow)**.** When an entry e is removed from a non-root-node with entry set L such that $size(L) \geq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor$ but $size(L \setminus \{e\}) < \lfloor \frac{3}{8} B_{\text{eff}} \rfloor$, 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 $size(R) \leq \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta$, the nodes can be merged to a single node $N := (L \setminus \{e\}) \cup R$:

$$size(N) = \overbrace{size(L \setminus \{e\})}^{\geq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} + \overbrace{size(R)}^{\geq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} \geq \lfloor \tfrac{1}{2} B_{\text{eff}} \rfloor$$
$$size(N) = \underbrace{size(L \setminus \{e\})}_{\leq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \delta} + \underbrace{size(R)}_{\leq \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta} \leq B_{\text{eff}}$$

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.

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

It follows that

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

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

$$size(L') \overset{\text{def}}{\geq} \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(L') = \underbrace{size(L \setminus \{e\})}_{= \lfloor \frac{3}{8} B_{\text{eff}} \rfloor - \delta} + \underbrace{size(S)}_{< \delta + \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} < \lfloor \tfrac{5}{8} B_{\text{eff}} \rfloor$$
$$size(R') = \overbrace{size(R)}^{> \lceil \frac{5}{8} B_{\text{eff}} \rceil + \delta} - \overbrace{size(S)}^{< \delta + \lfloor \frac{1}{4} B_{\text{eff}} \rfloor} > \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(R') < size(R) \leq B_{\text{eff}}$$

□

3 Algorithms

Procedure 3.1 (Insertion)**.** To insert a key-value-pair (k, v), in the first step the corresponding leaf node is searched (proc 3.3). Then the node increase handling (proc. 3.4) is performed.

Procedure 3.2 (Deletion)**.** To delete a key k, in the first step the corresponding leaf node is searched (proc 3.3). If such a leaf is found, the key-value-pair is removed and then the node decrease handling (proc. 3.5) is performed.

Procedure 3.3 (Search)**.** 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 3.4 (Node Increase Handling)**.** Input is a node N to which an element was inserted. Then, 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. 3.8).

ad 2. Redistribute $left(N)$ and N (proc. 3.7, redistribution cares about synchronization).

ad 3. Redistribute N and $right(N)$ (proc. 3.7, redistribution cares about synchronization).

ad 4. Split N to two nodes L and R (proc. 3.6). 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 *tree grows in height*). Otherwise, insert L into its parent and synchronize R with its parent (proc. 3.1 and proc. 3.8; insertion of L cares about its synchronization, note that the insertion of L into $parent(L)$ *might have changed* $parent(R)$).

Procedure 3.5 (Node Decrease Handling)**.** Input is a node N from which an element was deleted. Then, 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 *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. 3.8 and proc. 3.2; 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. 3.8).

ad 5: Redistribute N and $right(N)$ (proc. 3.7, redistribution cares about synchronization).

ad 6: Redistribute $left(N)$ and N (proc. 3.7, redistribution cares about synchronization).

Procedure 3.6 (Optimal Split of One Node)**.** Let $N = \{e_1, \ldots, e_d\}$ be the node that should be split to two nodes L and R. The goal is to calculate

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

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 3.7 (Node Redistribution)**.** Let L and R be two nodes. They are combined two a single node N (which is overflown). This is split optimally to L' and R' (proc. 3.6). 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. 3.8; note that the synchronization of L with $parent(L)$ *might have changed* $parent(R)$).

Procedure 3.8 (Node Synchronization)**.** Let N be a node and $P = parent(N)$ its parent. Then 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 $key(e_i) := \max N$, and
- set $count(e_i) := \sum_{e \in N} count(e)$.

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 $size(\max N) < size(key(e_i'))$, P might violate req. 2 after the update. A node decrease handling (proc. 3.5) of P must be performed.
- If $size(\max N) > size(key(e_i'))$, P might violate req. 1 after the update. A node increase handling (proc. 3.4) of P must be performed.
- Otherwise, the P must be written back and, if $i = deg(N)$ (and $key(e_i) \neq key(e_i')$) or if $count(e_i) \neq count(e_i')$, P must be synchronized with its parent (proc. 3.8).

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

One disadvantage of storing the indexes of the elements to offer access by the index number is that each insertion and each deletion needs to write all nodes along a path from the root to the leaf. The reorganization operations due to variable length keys even increase the number of nodes visited on the way from the leaf node back to the root.

This not only leads to many IO operations and thereby slows down insertion and deletion; it also raises problems with concurrency, because the root node becomes to a bottleneck that has to be locked for each operation.

Currently the concurrency issue is dealt with as follows: during an insertion or deletion other writers are locked out, but readers are still allowed. The operations have effect only on copies of the nodes in main memory. Before writing them back to disk, also concurrent readers are locked out.

Another way to deal with concurrency would be to slightly modify the B$^+$-Tree so that leaf nodes have no lower bound in size and apply locking as proposed in [4].

Obviously, the described B$^+$-Tree is only suitable for cases in which much more searches than insertion or deletions are performed. The Dingsbums 6 database system uses an Ada 95 implementation of this B$^+$-Tree as its core data structure.

References

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

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

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

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