Dingsbums B⁺-Tree

Christoph Schwering

April 6, 2009

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. 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 described B⁺-Tree supports bounded length keys and values and access by the index number of entries.

1 Definition

Definition 1.1 (Node)**.** A node $N = (M, E)$ of degree $d = deg(N)$ consists of some meta data M and an entry set $E = \{e_1, \dots, e_d\}$ of d entries.

The meta data stores the degree $deg(N)$ of the node and the left neighbor, the right neighbor and the parent, denoted by $left(N)$, $right(N)$, $parent(N)$ respectively. If such a neighbor or parent node does not exist, the 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 inner nodes $e = (k, v)$ if the context permits it. 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 $N = (M, E)$ in the tree:

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

Furthermore, the counts must be consistent in all states:

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

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 is not intended to support keys of a fixed size but *variable length keys*. (Note that the sizes of child addresses, count values and data values must be fixed, though.)

Hence, if $size(k)$ is the size of a certain key $k \in K$, we can uniquely determine the size of each entry e which will be denoted by $size(e)$. Then $size(E) = \sum_{e \in E} size(e)$ denotes the size of a complete entries $N = (M, E)$. Since the size of the meta data is fixed, it is easy to compute $size(N) = size(M) + size(E)$. Furthermore we will write $B_e = B - size(M)$. Finally, we require for each node $N = (M, E)$ two hold the requirements depicted in figure 1.

Remark 1.3 (Key size)*.* Note that req. 1 refers to a node element e which consists of can be either $e = (k, c, a)$ for a key k, a count value c and a child address a or $e = (k, v)$ for a key k and a value v. It follows that the maximum key size is

$$size(k) \leq \lfloor \tfrac{1}{n} B_e \rfloor - \max\{size(c) + size(a), size(v)\}.$$

This upper bound is statically determined, because the sizes of child addresses, count values and data values must be fixed.

2 Theorems

Theorem 2.1 (Node-Overflow)**.** A node $N' = (M, E \cup \{e\})$ with

$$size(E) \leq B_e \text{ but } size(E \cup \{e\}) > B_e$$

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

Proof. If $E \cup \{e\} = \{e_1, \dots, e_d\}$, set $L := \{e_1, \dots, e_i\}$ such that

$$size(L) \geq \lfloor \tfrac{1}{2} B_e \rfloor \text{ but } size(L \setminus \{e_i\}) < \lfloor \tfrac{1}{2} B_e \rfloor$$

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

$$size(L) \overset{\text{def}}{\geq} \lfloor \tfrac{1}{2} B_e \rfloor$$

$$size(L) < \lfloor \tfrac{1}{2} B_e \rfloor + \underbrace{size(e_i)}_{\leq \lfloor \frac{1}{n} B_e \rfloor} < \lfloor \tfrac{n+2}{2n} B_e \rfloor$$

$$size(R) = \overbrace{size(E \cup \{e\})}^{> B_e} - \overbrace{size(L)}^{< \lfloor \frac{n+2}{2n} B_e \rfloor} > \lceil \tfrac{n-2}{2n} B_e \rceil$$

$$size(R) = \underbrace{size(E \cup \{e\})}_{\leq B_e + \lfloor \frac{1}{n} B_e \rfloor} - \underbrace{size(L)}_{\geq \lfloor \frac{1}{2} B_e \rfloor} \leq \lceil \tfrac{1}{2} B_e \rceil + \lfloor \tfrac{1}{n} B_e \rfloor \leq \lceil \tfrac{n+2}{2n} B_e \rceil$$

By this, it directly follows that $\lfloor \frac{n-2}{2n} B_e \rfloor \leq size(L), size(R) \leq B_e$, 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 $size(L) \geq \lfloor \frac{n-2}{2n} B_e \rfloor$ but $size(L \setminus \{e\}) < \lfloor \frac{n-2}{2n} B_e \rfloor$, either entries from a neighbor can be moved or L can be merged with a neighbor.

$$size(e) \leq \left\lfloor \tfrac{1}{n} B_e \right\rfloor \quad \text{for all } e \in N \tag{1}$$
$$\left\lfloor \tfrac{n-2}{2n} B_e \right\rfloor \leq size(N) \leq B_e \quad \text{for all } N \text{ except the root} \tag{2}$$
$$0 \leq size(R) \leq B_e \quad \text{for the root node } R \tag{3}$$

Figure 1: Node filling requirements

Proof. Let R be the entry set of a neighbor of L. Depending on $size(R)$, either entries must be moved or nodes must be merged:

1. If $\left\lfloor \frac{n-2}{2n} B_e \right\rfloor \leq size(R) \leq \left\lfloor \frac{n+2}{2n} B_e \right\rfloor$, the nodes L and R can be merged to a node $N := (L \setminus \{e\}) \cup R$. The following equation shows that N fulfills req. 2:

$$\left\lfloor \tfrac{n-2}{2n} B_e \right\rfloor \leq size(N) = \underbrace{size(L \setminus \{e\})}_{< \left\lfloor \frac{n-2}{2n} B_e \right\rfloor} + \underbrace{size(R)}_{\leq \left\lfloor \frac{n+2}{2n} B_e \right\rfloor} \leq B_e.$$

2. If $size(R) > \left\lfloor \frac{n+2}{2n} B_e \right\rfloor$, one or more entries can be moved from R to L. We know that

$$size(L \setminus \{e\}) = \underbrace{size(L)}_{\geq \left\lfloor \frac{n-2}{2n} B_e \right\rfloor} - \underbrace{size(e)}_{\leq \left\lfloor \frac{1}{n} B_e \right\rfloor} \geq \left\lfloor \tfrac{n-4}{2n} B_e \right\rfloor.$$

Entries with a cumulated size $\geq size(e)$ must be moved from R to L. If $R = \{e_1, \ldots, e_n\}$, the set $S := \{e_1, \ldots, e_i\}$ such that

$$size(S) \geq size(e) \text{ but } size(S \setminus \{e_i\}) < size(e)$$

is the minimum set that must be moved from R to L to make L valid. In the worst case, $S \setminus \{e_i\}$ is verly large but not large enough, i.e. $size(S \setminus \{e_i\}) + \varepsilon = size(e)$, and the last entry e_i is unnecessarily large $size(e_i) = \left\lfloor \frac{1}{n} B_e \right\rfloor$. Hence, the size of S is bounded by

$$size(S) < 2 \cdot \left\lfloor \tfrac{1}{n} B_e \right\rfloor \leq \left\lfloor \tfrac{2}{n} B_e \right\rfloor.$$

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

$$size(L') = \overbrace{size(L \setminus \{e\})}^{< \left\lfloor \frac{n-2}{2n} B_e \right\rfloor} + \overbrace{size(S)}^{< \left\lfloor \frac{2}{2n} B_e \right\rfloor} < \left\lfloor \tfrac{n+2}{2n} B_e \right\rfloor$$
$$size(L') = size(L \setminus \{e\}) + size(S) \overset{\text{def}}{\geq} \left\lfloor \tfrac{n-2}{2n} B_e \right\rfloor$$
$$size(R') < size(R) \leq B_e$$
$$size(R') = \underbrace{size(R)}_{> \left\lfloor \frac{n+2}{2n} B_e \right\rfloor} - \underbrace{size(S)}_{\leq \left\lfloor \frac{n-4}{2n} B_e \right\rfloor} \geq \left\lfloor \tfrac{n-2}{2n} B_e \right\rfloor$$

By this, it directly follows that L' and R' fulfill req. 2.

□

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

Procedure 3.4 (Node Increase Handling)**.** Input is a node $N = (M, E)$ 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 $\min E$ to $left(N)$.

3: Try to shift the $\max E$ 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 $N = (M, E)$ 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 $E = \{e_1, \ldots, e_d\}$ be the entries of node $N = (M, E)$ that should be distribed on two nodes $L = (M_1, E_1)$ and $R = (M_2, E_2)$. The goal is to calculate

$$\arg\max_i \underbrace{\min_i\{size(\{e_1, \ldots, e_i\}), size(\{e_{i+1}, \ldots, e_d\})\}}_{=:s(i)}$$

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

Then set $E_1 := \{e_1, \ldots, e_i\}$ and $E_2 := \{e_{i+1}, \ldots, e_d\}$. Set $parent(L) := parent(R) := parent(N)$, $left(L) := left(N)$ $right(L) := R$, $left(R) := L$, $right(R) := right(N)$.

Proof. Initially, E_1 contains no elements. In the following iterations, in step 2, the $n+1$-th element is moved from E_2 to E_1 as long as this movement makes the smaller of the two nodes larger. □

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 $N = (M_N, E_N)$ be a node and $P = (M_P, E_P) = parent(N)$ its parent. Then let $e_i \in E_P$ 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 $count(e_i) := \sum_{e \in E_N} count(e)$.

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 the nodes from the root to the leaf. This raises problems with locking, because the root node becomes to a bottleneck that has to be locked for each operation. I do not know any good solution for this problem.

The Dingsbums 6 database system uses an Ada 95 implementation of this B$^+$-Tree as its core data structure. The problem of the root node being a bottleneck is not that important in this context since Dingsbums 6 is intended to be a read-optimized database system.

The definitions, proofs and procedures are hopefully correct. The B$^+$-Tree has tested with one billion entries, but there is still potential for errors.

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.