B-Trees

Andres Mendez-Vazquez

March 11, 2018

Contents

1 Introduction

As indexes in databases started to be counted in the millions, the data structures to represent those infixes required a way to have better bounds than the ones provided, for example, binary trees. In addition, the data structure could be so huge that it could not be stored in main memory. Therefore, it has been necessary to build a new type of data structure for handling these type of problems.

2 Motivation

Assuming that a disk spins at 10000 RPM, one revolution occurs in 0.006 of a second. Crudely speaking, one disk access (Moving around the disk) takes about the same time as 106 MIPS in a modern CPU/Core i7.

- Imagine that we decide to use a binary tree with 40 million of records. Then, under a balanced binary tree, we have $\lg 40,000,000 \approx 25$.

What if... instead of having two children, we decide to have 10 children instead... $\log_{10} 40000000 \approx 7.6$. We only require to waste 0.045 second or 805 MIPS. What if we use more children? A shorter three, but we still need to use a disk access per key!!! What if we add more keys into each node!!! After all if we use the idea of locality in a file you are looking for the next nearby.

3 B-Trees Data Structure

A B-tree T is a rooted tree with properties

- Every node x has the following values:
  - $x.n$= the number of keys stored at x
    - The keys are stored in nondecreasing order, $x.key_1 \leq x.key_2 \leq ... \leq x.key_n$.
    - $x.leaf$ a boolean value.
- Each internal node contains $x.n + 1$ pointers $x.c_1, x.c_2, ..., x.c_{x.n+1}$. Leaf nodes have no children.
  - the keys x.key_i separate ranges of keys: If k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq ... \leq x.key_{x.n} \leq k_{x.n+1}$$

- All leaves have the same depth h.
- Nodes have Upper and Lower Bounds. For this, we use an integer $t \geq 2$ (Minimum Degree).

– Every node other than the root must have $t-1$ keys $\Rightarrow$ every internal node $\neq$ root has t children i.e.
– Every node must contain at most $2t-1$ keys. Therefore, an internal node have at most $2t$ children. Full, if it contains exactly $2t-1$ keys

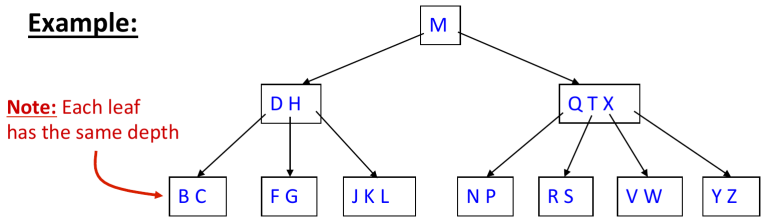

Figure 1: A B-Tree

3.1 The Height of the B-Tree

There is the following theorem to prove the height of the B-three.

Theorem 18.1

if $n \geq 1$, then for any n-key B-Tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.

Proof: In the slides, but look at (Fig. 2)

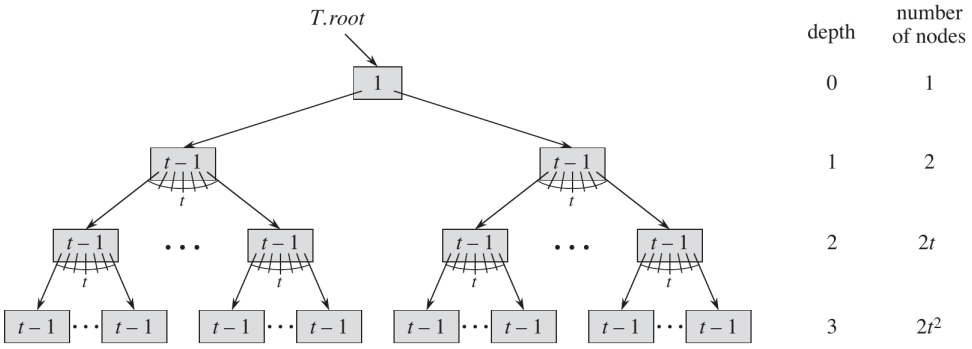

Figure 2: The Height of a B-Tree

3.2 How to Calculate the t Value when Page Size is a Given

It is possible to calculate the possible t value by assuming the following:

- For example, you have 4kb of page size
- Each key requires 32 bits = 4 bytes.
- Each pointers uses 32 bits = 4 bytes
- Each key in the tree has an associated payload of 64 bits = 8 bytes data structure.

Then, by using the following calculation:

$$\begin{aligned} 4 \cdot (2t-1) + 4 \cdot (2t) + 8 \cdot (2t-1) &\leq 4096 \\ 16t - 12 &\leq 4096 \\ t &\leq 255.25 \end{aligned}$$

Thus we can select $t = 255$ as a branching factor.

4 B-Trees Operations

For the following operations at the B-Tree data structure, we assume that:

1. The root of the B-tree is always in main memory:
   (a) Disk-Read are never performed on it.
   (b) Only When is written, we use a Disk-Write.
2. If a node is passed as parameter, it has already had all the necessary Disk-Read operations performed on it before hand.

Then, we have the following codes for each of the Basic Operations: Search and Creation.

4.1 Search

In this operation (Algorithm 1), we have the following costs:

- Worst Case:
  - $O(h) = O(\log_t n)$ disk reads when going through the entire tree.
  - $x.n < 2t \Rightarrow O(t)$ for searching the key at each node
  - Finally, we have that $O(th) = O(t \log_t n)$ CPU time.

Algorithm 1 The Search Operation in the B-Tree

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

4.2 Creating an Empty Tree

This is done by using the following algorithm (Algorithm 2). It requires $O(1)$ disk operation and $O(1)$ CPU time.

Algorithm 2 The Create Operation in the B-Tree

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

4.3 Insertion Operation

Here is where the things become interesting!!! Why? Because we need to take in account the following.

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays.

This means that if a node has 2t-1 keys something needs to be done in order to make space in the node. For this, we follow the next operations:

1. Split the node around the median key.
2. You finish with two nodes of size $t-1$ and the median key y.

3. Promote the median key to the father node to identify the new ranges.

4. If the father is full recursively split the father to make room.

The code for the operations Insertion and Split is in the slides. It works as follow:

1. Insert at the root of T and the key k to be inserted.

   (a) Use a a temporary variable r to look at the root

   (b) If $r.n == 2t - 1$ Then prepare to split by creating an alternate s father node (Look at the Slides).

      i. Then Split the node s using Split-Child

      ii. Inger using the Insert-Non full operation.

   (c) else Insert using the Insert-Non full operation.

2. Split-Child using a non-full node x (Assumed to be in main memory), an index i such that $y = x.c_i$ is a full child (Again assumed to be in main memory).

   (a) The code works as follow the element y has $2t$ children ($2t - 1$ keys) but is reduced to t children. For this, the new node z takes the t largest children from y, and z becomes a new child of x. The code's lines work as follow:

      i. Lines 1-11 creates node z and it gives the largest $t - 1$ keys and corresponding t children of y.

      ii. Lines 12-21 insert z as child of x.

         A. In lines 13-16 space is made in the array of pointers to insert the pointer $x.c_{i+1}$ toward z.

         B. In lines 18-20 space is made in the array of keys to insert the key $y.key_t$ (The one being promoted toward x when y is splitted) in x.

      iii. Last lines writes everything into the disk.

3. Insert-Non full using a node x to insert into the key k. The code's lines work as follow:

   (a) Lines 2-9 deals with the case that x is a leaf (Do not worry the codes before entering these lines will never allow to insert into a full node).

   (b) Line 11-21 deals with the case of x is not a leaf:

      i. Lines 11-14 gets the correct child to insert the key k.

      ii. Then $x.c_i$ is read into main memory.

      iii. Line 15 checks if the child is full:

         A. If that happens the child is split!!!

         B. In line 17 the code checks where the key should be in y or z.

      iv. Finally in line 21 the key k is inserted into $x.c_i$.

4.4 Deletion of a key

The main idea is to descend through the B-Tree to ensure that any not x that is considered has at least t keys. For this, it could be necessary to move a key down from parent. The code for deletion works as follow:

1. Empty root - make root s only child the new root.

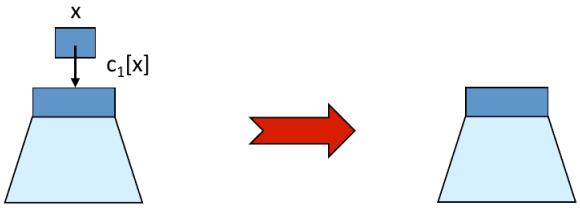

Figure 3: Remove Root.

2. If the key k is in node x and x is a leaf, delete the key k from x.

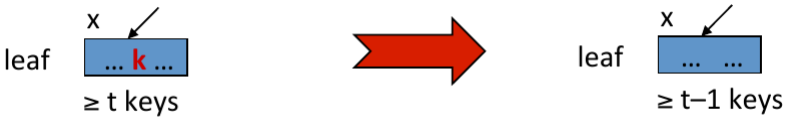

Figure 4: Deleting x from leaf.

3. If the key k is in node x and x is an internal node, do the following:

   (a) If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' , and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

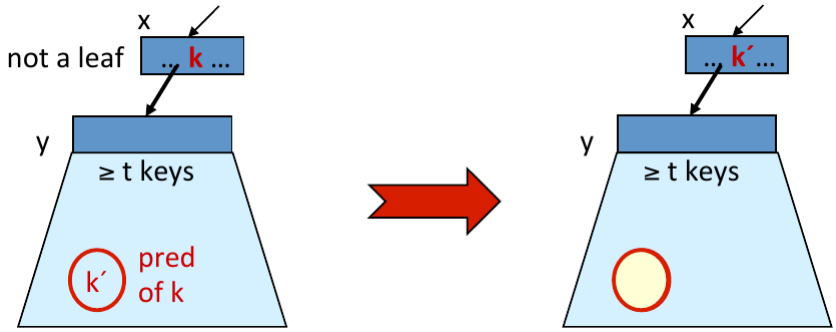

Figure 5: Deleting x from internal node, case 2.a.

(b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

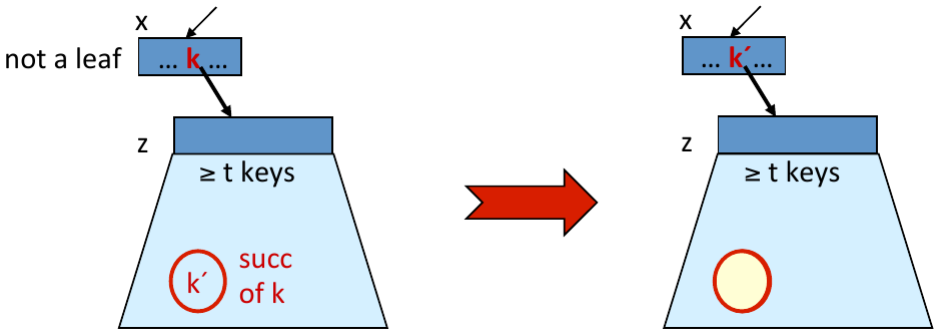

Figure 6: Deleting x from internal node, case 2.b.

(c) Otherwise, if both y and z have only $t-1$ keys, merge k and all of into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.

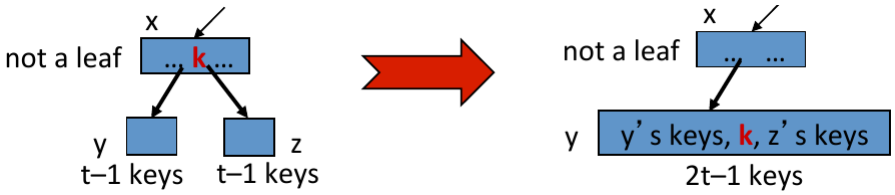

Figure 7: Deleting x from internal node, case 2.c.

4. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step $3a$ or $3b$ as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

   (a) If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

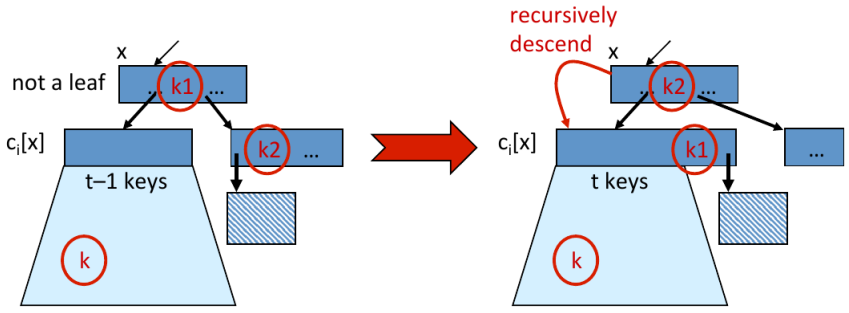

Figure 8: Deleting x from internal node, case 3.a.

   (b) If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

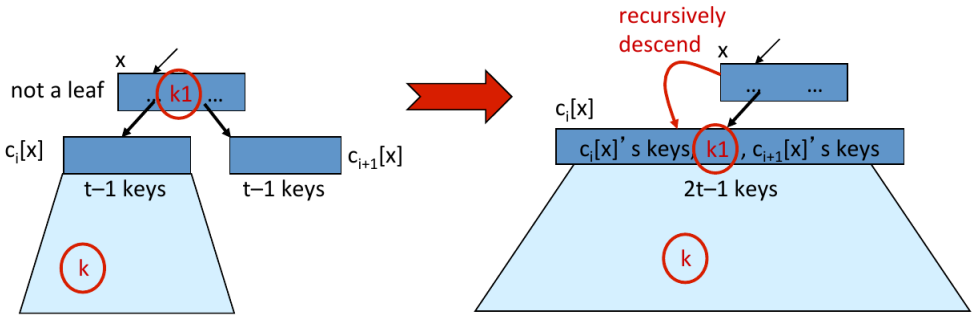

Figure 9: Deleting x from internal node, case 3.b.

APPENDIX - PSEUDO-CODE for Delete

Algorithm 3 B_Tree_Delete

```
void B_Tree_Delete ( Node root, Item k )
-- pre: root node of B-tree
-- post: if k in B-tree, remove corresponding node

   remove( root, k )

   if root ≠ NIL and root.n = 0 then
      root ← root.c0
      Disk_Write ( root )
```

Algorithm 4 remove

```
void remove (Node x, Item k)
-- pre: x points to root node of subtree
-- post: if k in subtree, remove corresponding node from subtree

  i ← x.n
  while i ≥ 1  and  k < x.key_i do     -- search for k
       i ← i - 1

  if i ≥ 1 and k ≠ x.key_i then        -- not found, check child
     Disk_Read ( x.c_i)
     remove( x.c_i, k )
  else                                  -- found

     if x.leaf then                     -- at a leaf, delete k
        while i < x.n do                  -- copy over k
           x.key_i ← x.key_i+1
           i ← i + 1
        x.n ← x.n - 1
     else                                  -- not at a leaf, find pointer to correct
        copy_predecessor( x, i)          -- subtree and recurse
        Disk_Read ( x.c_i )
        remove( x.c_i , x.key_i )

  if not x.leaf and x.c_i.n < t - 1 then
       restore( x, i )                   -- child < minimum degree, restore

  Disk_Write (x)
```

Algorithm 5 copy_predecessor

```
void copy_predecessor (Node x, int i)
-- pre: x points to non-leaf node with entry at i
-- post: if k in subtree, remove corresponding node from subtree

  leaf ← x.c_i                          -- x.c_i is node to the left of x

  while leaf.c_leaf.n ≠ NIL do          -- Go as far right to a leaf
      leaf ← leaf.c_leaf.n
      Disk_Read ( leaf )

  x.key_i ← leaf.key_leaf.n-1           -- Copy predecessor key

  Disk_Write (x)
```

Algorithm 6 Restore

```
void restore (Node x, int i)
-- pre: x points to non-leaf node with entry at x.c_i has one too few entries
-- post: An entry is taken from elsewhere to restore minimum number
--        of entries in the node to which x.c_i points

  if i = x.n then                       -- rightmost child
     if x.c_i-1.n > t - 1 then
      move_right( x, i - 1)
     else
      combine( x, i )
  else
       if i = 1 then                    -- leftmost child
          if x.c_2.n > t - 1 then
             move_left( x, 2 )
           else
             combine( x, 2 )
        else                            -- remaining cases: intermediate branches
          if x.c_i-1.n > t - 1 then
            move_right( x, i-1 )
          else
            if x.c_i+1.n > t - 1 then
               move_left(x, i+1)
            else
               combine( x, i )
```

Algorithm 7 move_left

```
void move_left (Node x, int i)
-- pre: x points to node with more than minimum number of entries in child i
--        and one too few in child i-1
-- post: leftmost entry of child i is moved to x, which has sent an entry to child i-1

  left ← x.c_{i-1}
  right ← x.c_i

  left.key_{left.n} ← x.key_{i-1}          -- take parent entry

  left.n ←  left.n + 1

  left.c_{left.n} ← right.c_1

  left.key_{i-1} ← right.key_1             -- add right to parent

  right.n ← right.n - 1

  for j ← 1 to right.n-1 do                -- move right entries to fill hole
      right.key_j ← right.key_{j+1}
      right.c_j ← right.c_{j+1}

   right.c_{right.n} ← right.c_{right.n+1}

   Disk_Write( left )
   Disk_Write( right )
```

Algorithm 8 move_right

```
void move_right (Node x, int i)
-- pre: x points to node with more than minimum number of entries in child i
--        and one too few in child i+1
-- post: rightmost entry of child i is moved to x, which has sent an entry to child i+1

  left ← x.c_i
  right ← x.c_{i+1}

  right.c_{right.n+1} ← right.c_{right.n}

  for j ← right.n downto 1 do              -- make room for new entry
      right.key_j ← right.key_{j-1}
      right.c_j ← right.c_{j-1}

  right.n ← right.n + 1

  right.c_1 ← left.c_{left.n}

   left.n ←  left.n - 1

  x.key_i ← left.key_{left.n}

   Disk_Write( x )
   Disk_Write( left )
   Disk_Write( right )
```

Algorithm 9 combine

```
void combine (Node x, int i)
-- pre: x points to a node with child i and i-1 with too few entries to move
-- post: nodes at i and i-1 have been combined into one node

  left ← x.c_{i-1}
  right ← x.c_i

  left.key_{left.n} ← x.key_{i-1}

   left.n ← left.n + 1

  left.c_{left.n} ← right.c_1

  for j ← 1 to right.n-1 do
       left.key_{left.n} ← right.key_j
       left.n ←  left.n + 1
       left.c_{left.n} ← right.c_{j+1}

  x.n ← x.n - 1

  for j ← i-1 to x.n-1 do
       x.key_j ← x.key_{j+1}
       x.c_{i+1} ← x.c_{i+2}

   right = NIL

   Disk_Write( x )
   Disk_Write( left )
   Disk_Write( right )
```