B-trees

Andreas Kaltenbrunner, Lefteris Kellis & Dani Martí
What are B-trees?

• B-trees are **balanced search trees**: height = $O(\log(n))$ for the worst case.
• They were designed to work well on **Direct Access secondary storage devices** (magnetic disks).
• Similar to **red-black** trees, but show **better performance on disk I/O operations**.
• B-trees (and variants like **B+** and **B**$^*$ trees) are widely used in **database systems**.
Motivation

Data structures on secondary storage:

- Memory capacity in a computer system consists broadly on 2 parts:
  1. **Primary memory**: uses memory chips.
  2. **Secondary storage**: based on magnetic disks.
- Magnetic disks are **cheaper** and have **higher capacity**.
- But they are much slower because they have moving parts.

B-trees try to read **as much information as possible in every disk access** operation.
The 21 English consonants as keys of a B-tree:

- Every internal node $x$ containing $n[x]$ keys has $n[x] + 1$ children.
- All leaves are at the same depth in the tree.
B-tree: definition

A **B-tree** $T$ is a rooted tree (with root $\text{root}[T]$) with properties:

- Every node $x$ has **four** fields:
  1. The number of keys currently stored in node $x$, $n[x]$.
  2. The $n[x]$ keys themselves, stored in nondecreasing order:
     \[
     \text{key}_1[x] \leq \text{key}_2[x] \leq \cdots \leq \text{key}_{n[x]}[x].
     \]
  3. A boolean value,
     \[
     \text{leaf}[x] = \begin{cases} 
     \text{True} & \text{if } x \text{ is a leaf}, \\
     \text{False} & \text{if } x \text{ is an internal node}.
     \end{cases}
     \]
  4. $n[x] + 1$ pointers, $c_1[x]$, $c_2[x]$, \ldots, $c_{n[x]+1}[x]$ to its children.
     (As leaf nodes have no children their $c_i$ are undefined).

- Representing pointers and keys in a node:
B-tree: definition (II)

Properties (cont):

• The keys $\text{key}_i[x]$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $c_i[x]$, then:

\[ k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \leq \ldots \leq \text{key}_n[x] \leq k_n[x]+1. \]

• All leaves have the **same height**, which is the tree’s height $h$.

• There are upper or lower bounds on the number of keys on a node. To specify these bounds we use a fixed integer $t \geq 2$, the **minimum degree** of the B-tree:

  – **lower bound**: every node other than root must have at least $t - 1$ keys $\implies$ At least $\lceil t \rceil$ children.

  – **upper bound**: every node can contain at most $2t - 1$ keys $\implies$ every internal node has at most $2t$ children.
The height of a B-tree (I)

Example (worst-case): A B-tree of height 3 containing a minimum possible number of keys.

Inside each node $x$, we show the number of keys $n[x]$ contained.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$2t$</td>
</tr>
<tr>
<td>3</td>
<td>$2t^2$</td>
</tr>
</tbody>
</table>
The height of a B-tree (II)

• Number of disk accesses proportional to the height of the B-tree.
• The worst-case height of a B-tree is
  \[ h \leq \log_t \frac{n + 1}{2} \sim O(\log_t n) \, . \]
• Main advantage of B-trees compared to red-black trees:
  The base of the logarithm, \( t \), can be much larger.
  \( \implies \) B-trees save a factor \( \sim \log t \) over red-black trees in the number of nodes examined in tree operations.
  \( \implies \) Number of disk accesses substantially reduced.
Basic operations on B-trees

Details of the following operations:

- **B-Tree-Search**
- **B-Tree-Create**
- **B-Tree-Insert**
- **B-Tree-Delete**

Conventions:

- Root of B-tree is always in main memory (**DISK-READ** on the root is never required)
- Any node passed as parameter must have had a **DISK-READ** operation performed on them.

Procedures presented are all **top down algorithms** (no need to back up) starting at the root of the tree.
 Searching a B-tree (I)

2 inputs: \texttt{x}, \textbf{pointer} to the root node of a subtree, 
\( k \), a \textbf{key} to be searched in that subtree.

\begin{align*}
\text{function } \text{B-Tree-Search}(x, k) \text{ returns } (y, i) \text{ such that } \text{key}_i[y] = k \text{ or NIL} \\
i \leftarrow 1 \\
\text{while } i \leq n[x] \text{ and } k > \text{key}_i[x] \\
\text{do } i \leftarrow i + 1 \\
\text{if } i \leq n[x] \text{ and } k = \text{key}_i[x] \\
\text{then return } (x, i) \\
\text{if } \text{leaf}[x] \\
\text{then return } \text{NIL} \\
\text{else } \text{Disk-Read}(c_i[x]) \\
\text{return } \text{B-Tree-Search}(c_i[x], k)
\end{align*}

At each internal node \texttt{x} we make an \((n[x] + 1)\)-way branching decision.
Searching a B-tree (II)

- Number of disk pages accessed by B-TREE-SEARCH
  \[ \Theta(h) = \Theta(\log_t n) \]

- Time of \textbf{while} loop within each node is \( O(t) \) therefore the total CPU time
  \[ O(th) = O(t \log_t n) \]
Creating an empty B-tree

\begin{algorithm}
\textbf{B-Tree-Create}(T)
\begin{algorithmic}
\STATE $x \leftarrow \text{Allocate-Node}()$
\STATE $\text{leaf}[x] \leftarrow \text{true}$
\STATE $n[x] \leftarrow 0$
\STATE $\text{Disk-Write}(x)$
\STATE $\text{root}[T] \leftarrow x$
\end{algorithmic}
\end{algorithm}

- \text{Allocate-Node}() allocates one disk page to be used as a new node
- requires $O(1)$ disk operations an $O(1)$ CPU time
• Inserting a key into a B-tree is more complicated than in binary search tree.
• **Splitting** of a full node $y$ ($2t - 1$ keys) fundamental operation during insertion.
• Splitting around **median key** $key_t[y]$ into 2 nodes.
• Median key moves up into $y$’s parent (which has to be nonfull).
• If $y$ is root node tree height grows by 1.
Splitting a node in a B-tree (II)

3 inputs: $x$, a **nonfull** internal node,

$i$, an index,

$y$, a node such that $y = c_i[x]$ is a **full** child of $x$.

```
B-Tree-Split-Child(x, i, y)
  z ← Allocate-Node()
  leaf[z] ← leaf[y]
  n[z] ← t − 1
  for j ← 1 to t − 1
    do key_j[z] ← key_{j+t}[y]
    if not leaf[y]
      then for j ← 1 to t
        do c_j[z] ← c_{j+t}[y]
  n[y] ← t − 1

for j ← n[x] + 1 downto i + 1
  do c_{j+1}[x] ← c_j[x]
  c_{i+1}[x] ← z
  for j ← n[x] downto i
    do key_{j+1}[x] ← key_j[x]
  key_i[x] ← key_i[y]
  n[x] ← n[x] + 1
  Disk-Write(y)
  Disk-Write(z)
  Disk-Write(x)
```

CPU time used by $\text{B-Tree-Split-Child}$ is $Θ(t)$ due to the loops

$B$-trees, A. Kaltenbrunner, L. Kellis & D. Martí 14
Inserting a key into a B-tree (I)

- The key is always inserted in a leaf node
- Inserting is done in a single pass down the tree
- Requires $O(h) = O(\log_t n)$ disk accesses
- Requires $O(th) = O(t \log_t n)$ CPU time
- Uses B-Tree-Split-Child to guarantee that recursion never descends to a full node
Inserting a key into a B-tree (II)

2 inputs: $T$, the root node,
$k$, key to insert.

\[
\begin{align*}
\text{B-Tree-Insert}(T, k) \\
r & \leftarrow \text{root}[T] \\
\text{if } n[r] = 2t - 1 & \text{ then } s \leftarrow \text{Allocate-Node()} \\
& \quad \text{root}[T] \leftarrow s \\
& \quad \text{leaf}[s] \leftarrow \text{FALSE} \\
& \quad n[s] \leftarrow 0 \\
& \quad c_1[s] \leftarrow r \\
& \quad \text{B-Tree-Split-Child}(s, 1, r) \\
& \quad \text{B-Tree-Insert-Nonfull}(s, k) \\
\text{else } & \quad \text{B-Tree-Insert-Nonfull}(r, k)
\end{align*}
\]

Uses $\text{B-Tree-Insert-Nonfull}$ to insert key $k$ into nonfull node $x$
Inserting a key into a nonfull node of a B-tree

**B-Tree-Insert-Nonfull**(\(x, k\))

\[
i \leftarrow n[x]
\]

**if** leaf\([x]\)

**then** while \(i \geq 1\) and \(k < \text{key}_i[x]\)

**do** key\(_{i+1}[x] \leftarrow \text{key}_i[x]\

\[
i \leftarrow i - 1
\]

key\(_{i+1}[x] \leftarrow k\)

\[
n[x] \leftarrow n[x] + 1
\]

**Disk-Write**(\(x\))

**else** while \(i \geq 1\) and \(k < \text{key}_i[x]\)

**do** \(i \leftarrow i - 1\

\[
i \leftarrow i + 1
\]

**Disk-Read**(\(c_i[x]\))

**if** \(n[c_i[x]] = 2t - 1\)

**then** **B-Tree-Split-Child**(\(x, i, c_i[x]\))

**if** \(k > \text{key}_i[x]\)

**then** \(i \leftarrow i + 1\

**B-Tree-Insert-Nonfull**(\(c_i[x], k\))

---

*B-trees, A. Kaltenbrunner, L. Kellis & D. Martí* 17
Inserting a key - Examples (I)

Initial tree:

\[ t = 3 \]

2 inserted:

17 inserted:

(to the previous one)
Inserting a key - Examples (II)

Initial tree:
\[ t = 3 \]

\[
\begin{array}{c}
1 2 3 4 5 \\
10 11 \\
14 15 \\
17 18 19 \\
21 22 \\
25 26 \\
\end{array}
\]

12 inserted:

\[
\begin{array}{c}
1 2 3 4 5 \\
10 11 12 \\
14 15 \\
17 18 19 \\
21 22 \\
25 26 \\
\end{array}
\]

6 inserted:

\[
\begin{array}{c}
1 2 \\
4 5 6 \\
10 11 12 \\
14 15 \\
17 18 19 \\
21 22 \\
25 26 \\
\end{array}
\]

(to the previous one)
Deleting a Key from a B-tree

- Similar to insertion, with the addition of a couple of special cases
- Key can be deleted from any node.
- More complicated procedure, but similar performance figures: $O(h)$ disk accesses, $O(th) = O(t \log_t n)$ CPU time
- Deleting is done in a single pass down the tree, but needs to return to the node with the deleted key if it is an internal node
- In the latter case, the key is first moved down to a leaf. Final deletion always takes place on a leaf
Deleting a Key — Cases I

- Considering 3 distinct cases for deletion
- Let $k$ be the key to be deleted, $x$ the node containing the key. Then the cases are:

1. If key $k$ is in node $x$ and $x$ is a leaf, simply delete $k$ from $x$

2. If key $k$ is in node $x$ and $x$ is an internal node, there are three cases to consider:
   (a) If the child $y$ that precedes $k$ in node $x$ has at least $t$ keys (more than the minimum), then find the predecessor key $k'$ in the subtree rooted at $y$. Recursively delete $k'$ and replace $k$ with $k'$ in $x$
   (b) Symmetrically, if the child $z$ that follows $k$ in node $x$ has at least $t$ keys, find the successor $k'$ and delete and replace as before. Note that finding $k'$ and deleting it can be performed in a single downward pass
   (c) Otherwise, if both $y$ and $z$ have only $t - 1$ (minimum number) keys, merge $k$ and all of $z$ into $y$, so that both $k$ and the pointer to $z$ are removed from $x$. $y$ now contains $2t - 1$ keys, and subsequently $k$ is deleted
3. If key $k$ is not present in an internal node $x$, determine the root of the appropriate subtree that must contain $k$. If the root has only $t - 1$ keys, execute either of the following two cases to ensure that we descend to a node containing at least $t$ keys. Finally, recurse to the appropriate child of $x$

(a) If the root has only $t - 1$ keys but has a sibling with $t$ keys, give the root an extra key by moving a key from $x$ to the root, moving a key from the roots immediate left or right sibling up into $x$, and moving the appropriate child from the sibling to $x$

(b) If the root and all of its siblings have $t - 1$ keys, merge the root with one sibling. This involves moving a key down from $x$ into the new merged node to become the median key for that node.
Deleting a Key — Case 1

The first and simple case involves deleting the key from the leaf. $t - 1$ keys remain.
Deleting a Key — Cases 2a, 2b

Initial tree:

13 deleted:

- Case 2a is illustrated. The predecessor of 13, which lies in the preceding child of $x$, is moved up and takes 13's position. The preceding child had a key to spare in this case.
Deleting a Key — Case 2c

Initial tree:

```
  16
  /  
3 7 12 /      \ 20 23
 | \   |      |
1 2 4 5 10 11 14 15 17 18 19 21 22 24 26
```

7 deleted:

```
  16
  /  
3 12 /      \ 20 23
 | \   |      |
1 2 4 5 10 11 14 15 17 18 19 21 22 24 26
```

• Here, both the preceding and successor children have $t - 1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from that leaf.
Deleting a Key — Case 3b

Initial tree:
Key 4 to be deleted

- The catchy part. Recursion cannot descend to node 3, 12 because it has $t - 1$ keys. In case the two leaves to the left and right had more than $t - 1$, 3, 12 could take one and 3 would be moved down.
- Also, the sibling of 3, 12 has also $t - 1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root.
- Therefore the root has to be pushed down merging its two children, so that 4 can be safely deleted from the leaf.
Deleting a Key — Case 3b II

Initial tree:

4 deleted:

Outcome:
Deleting a Key — Case 3a

Initial tree:

2 deleted:

(To the previous one)

- In this case, 1, 2 has \( t - 1 \) keys, but the sibling to the right has \( t \). Recursion moves 5 to fill 3's position, 5 is moved to the appropriate leaf, and deleted from there.
Deletion of a Key — Pseudo Code I

\begin{verbatim}
B-Tree-Delete-Key(x, k)
    if not leaf[x] then
        y ← Preceding-Child(x)
        z ← Successor-Child(x)
        if n[y] > t - 1 then
            \( k' ← \text{Find-Predecessor-Key}(k, x) \)
            \( \text{Move-Key}(k', y, x) \)
            \( \text{Move-Key}(k, x, z) \)
            B-Tree-Delete-Key(k, z)
        else if n[z] > t - 1 then
            \( k' ← \text{Find-Successor-Key}(k, x) \)
            \( \text{Move-Key}(k', z, x) \)
            \( \text{Move-Key}(k, x, y) \)
            B-Tree-Delete-Key(k, y)
        else
            \( \text{Move-Key}(k, x, y) \)
            \( \text{Merge-Nodes}(y, z) \)
            B-Tree-Delete-Key(k, y)
    \end{verbatim}
Deleting a Key — Pseudo Code II

else (leaf node)
    \( y \leftarrow \text{Preceding-Child}(x) \)
    \( z \leftarrow \text{Successor-Child}(x) \)
    \( w \leftarrow \text{root}(x) \)
    \( v \leftarrow \text{RootKey}(x) \)
    if \( n[x] > t - 1 \) then \( \text{REMOVE-KEY}(k, x) \)
    else if \( n[y] > t - 1 \) then
        \( k' \leftarrow \text{Find-Predecessor-Key}(w, v) \)
        \( \text{MOVE-KEY}(k', y, w) \)
        \( k' \leftarrow \text{Find-Successor-Key}(w, v) \)
        \( \text{MOVE-KEY}(k', w, x) \)
        \( \text{B-TREE-DELETE-KEY}(k, x) \)
    else if \( n[w] > t - 1 \) then
        \( k' \leftarrow \text{Find-Successor-Key}(w, v) \)
        \( \text{MOVE-KEY}(k', z, w) \)
        \( k' \leftarrow \text{Find-Predecessor-Key}(w, v) \)
        \( \text{MOVE-KEY}(k', w, x) \)
        \( \text{B-TREE-DELETE-KEY}(k, x) \)
Deleting a Key — Pseudo Code III

```
else
    s ← FIND-SIBLING(w)
    w' ← root(w)
    if n[w'] = t − 1 then
        MERGE-NODES(w', w)
        MERGE-NODES(w, s)
        B-TREE-DELETE-KEY(k, x)
    else
        MOVE-KEY(v, w, x)
        B-TREE-DELETE-KEY(k, x)
```

- **Preceding-Child(x)** Returns the left child of key $x$.
- **Move-Key(k, n_1, n_2)** Moves key $k$ from node $n_1$ to node $n_2$.
- **Merge-Nodes(n_1, n_2)** Merges the keys of nodes $n_1$ and $n_2$ into a new node.
- **Find-Predecessor-Key(n, k)** Returns the key preceding key $k$ in the child of node $n$.
- **Remove-Key(k, n)** Deletes key $k$ from node $n$. $n$ must be a leaf node.