CS143: B+Tree

Professor Junghoo “John” Cho
B+Tree

• Most popular index structure in RDBMS

• Advantage
  • Suitable for dynamic updates
  • Balanced
  • Minimum space usage guarantee

• Disadvantage
  • Non-sequential index blocks
B+Tree (n=3)

- n: # of pointer spaces in a node
- Balanced: All leaf nodes are at the same level

<table>
<thead>
<tr>
<th>20</th>
<th>Susan</th>
<th>2.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>James</td>
<td>3.6</td>
</tr>
<tr>
<td>50</td>
<td>Peter</td>
<td>1.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Leaf Node (n=3)

- All pointers (except the last one) point to tuples
- At least half of the pointer spaces are used.
  (more precisely, \(\lceil(n + 1)/2\rceil\) pointers)
Non-leaf Node (n=3)

- Points to the nodes one-level below
  - No direct pointers to tuples
- At least half of the pointer spaces used (precisely, \( \lceil n/2 \rceil \))
  - except root, where at least 2 pointer spaces used
Space Usage Guarantee

• B+Tree nodes have at least
  • Leaf (non-root): \(\lceil(n + 1)/2\rceil\) pointers, \(\lceil(n + 1)/2\rceil - 1\) keys
  • Non-leaf (non-root): \(\lceil n/2 \rceil\) pointers, \(\lceil n/2 \rceil - 1\) keys
  • Root: 2 pointers, 1 key

<table>
<thead>
<tr>
<th>n=4</th>
<th>Minimum</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf</td>
<td><img src="image" alt="Leaf diagram" /></td>
<td><img src="image" alt="Full diagram" /></td>
</tr>
<tr>
<td>Non-leaf</td>
<td><img src="image" alt="Non-leaf diagram" /></td>
<td><img src="image" alt="Full diagram" /></td>
</tr>
</tbody>
</table>

\[
\lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil = 3 \quad \lceil \frac{5}{2} \rceil - 1 = 2
\]
Search on B+tree

- Find 30, 60, 70?
- Find a greater key and follow the link on the left
  (Algorithm: Figure 14.11 on textbook)
B+Tree Insertion

1. no overflow
2. leaf overflow
3. non-leaf overflow
4. new root
1. No Overflow

- Insert 60
2. Leaf Overflow

• Insert 55

• No space to store 55
2. Leaf Overflow

- Insert 55

Split the leaf into two. Put the keys half and half
2. Leaf Overflow

• Insert 55

• Split the leaf into two. Put the keys half and half
2. Leaf Overflow

- Insert 55

- *Copy* the first key of the new node to parent
2. Leaf Overflow

• Insert 55

• Q: After split, leaf nodes always half full?
3. Non-leaf Overflow

• Insert 52

Leaf overflow. Split and copy the first key of the new node
3. Non-leaf Overflow

- Insert 52
3. Non-leaf Overflow

• Insert 52
3. Non-leaf Overflow

• Insert 52
3. Non-leaf Overflow

- Insert 52

Split the node into two. *Move* up the key in the middle.
3. Non-leaf Overflow

- Insert 52
3. Non-leaf Overflow

- Insert 52

Q: After split, non-leaf at least half full?
4. New Root

- Insert 25
4. New Root

- Insert 25

Overflow!
4. New Root

• Insert 25
4. New Root

• Insert 25
• Q: At least 2 pointers at root?
B+Tree Insertion

• Leaf node overflow
  • The first key of the new node is *copied* to the parent

• Non-leaf node overflow
  • The middle key is *moved* to the parent

• Detailed algorithm: Figure 14.17
B+Tree Deletion

1. No underflow
2. Leaf underflow (coalesce with neighbor)
3. Leaf underflow (redistribute with neighbor)
4. Non-leaf underflow (coalesce with neighbor)
5. Non-leaf underflow (redistribute with neighbor)
6. Tree depth reduction

In the examples, \( n = 4 \)

- Underflow for non-leaf when fewer than \( \lceil n/2 \rceil = 2 \) pointers
- Underflow for leaf when fewer than \( \lceil (n + 1)/2 \rceil = 3 \) pointers
- Nodes are labeled as \( a, b, c, d, \ldots \)
1. No Underflow

- Delete 25
1. No Underflow

- Delete 25
  - Underflow? Min 3 ptrs. Currently 3 ptrs
2. Coalesce Leaf with Neighbor

- Delete 50
2. Coalesce Leaf with Neighbor

- Delete 50
  - Underflow? Min 3 ptrs, currently 2.
2. Coalesce Leaf with Neighbor

- Delete 50
  - Try to merge with a sibling

Can be merged?

underflow!
2. Coalesce Leaf with Neighbor

• Delete 50
  • Merge $c$ and $d$. Move everything on the right to the left.
2. Coalesce Leaf with Neighbor

- Delete 50
  - Once everything is moved, delete $d$
2. Coalesce Leaf with Neighbor

- Delete 50
  - After leaf node merge,
    - From its parent, delete the pointer and key to the deleted node
2. Coalesce Leaf with Neighbor

- Delete 50
  - Check underflow at $a$. Min 2 ptrs, currently 3
3. Redistribute Leaf with Neighbor

- Delete 50
3. Redistribute Leaf with Neighbor

• Delete 50
  • Underflow? Min 3 ptrs, currently 2
  • Check if $d$ can be merged with its sibling $c$ or $e$
  • If not, redistribute the keys in $d$ with a sibling
    • Say, with $c$
3. Redistribute Leaf with Neighbor

- Delete 50
  - Redistribute c and d, so that nodes c and d are roughly “half full”
    - Move the key 30 and its tuple pointer to the d
3. Redistribute Leaf with Neighbor

- Delete 50
  - Update the key in the parent
3. Redistribute Leaf with Neighbor

• Delete 50
  • No underflow at $a$. Done.
4. Coalesce Non-Leaf with Neighbor

- Delete 20
  - Underflow! Merge \(d\) with \(e\).
    - Move everything in the right to the left
4. Coalesce Non-Leaf with Neighbor

- Delete 20
  - From the parent node, delete pointer and key to the deleted node
4. Coalesce Non-Leaf with Neighbor

- **Delete 20**
  - Underflow at $b$? Min 2 ptrs, currently 1.
  - Try to merge with its sibling.
    - Nodes $b$ and $c$: 3 ptrs in total. Max 4 ptrs.
    - Merge $b$ and $c$. 
4. Coalesce Non-Leaf with Neighbor

- **Delete 20**
  - Merge $b$ and $c$
    - Pull down the mid-key 50 in the parent node
    - Move everything in the right node to the left.

- **Very important:** when we merge *non-leaf nodes*, we always pull down the mid-key in the parent and place it in the merged node.
4. Coalesce Non-Leaf with Neighbor

- Delete 20
  - Merge $b$ and $c$
    - Pull down the mid-key 50 in the parent node
    - Move everything in the right node to the left.

- Very important: when we merge *non-leaf nodes*, we always pull down the mid-key in the parent and place it in the merged node.
4. Coalesce Non-Leaf with Neighbor

- Delete 20
  - Delete pointer to the merged node.
4. Coalesce Non-Leaf with Neighbor

- Delete 20
5. Redistribute Non-Leaf with Neighbor

- Delete 20
  - Underflow! Merge $d$ with $e$. 

```
  a: [50, 99]
  b: [30]
  c: [70, 90, 97]
  d: [10, 20]
  e: [30, 40]
  f: [50, 60]
  g: [70]
```
5. Redistribute Non-Leaf with Neighbor

- Delete 20
  - After merge, remove the key and ptr to the deleted node from the parent
5. Redistribute Non-Leaf with Neighbor

- Delete 20
  - Underflow at $b$? Min 2 ptrs, currently 1.
  - Merge $b$ with $c$? Max 4 ptrs, 5 ptrs in total.
  - If cannot be merged, redistribute the keys with a sibling.
    - Redistribute $b$ and $c$
Redistribution at a non-leaf node is done in two steps.

Step 1: Temporarily, make the left node $b$ “overflow” by pulling down the mid-key and moving everything to the left.
5. Redistribute Non-Leaf with Neighbor

- Delete 20

Step 2: Apply the "overflow handling algorithm" (the same algorithm used for B+tree insertion) to the overflowed node
  - Detailed algorithm in the next slide
5. Redistribute Non-Leaf with Neighbor

- Delete 20

**Step 2: “overflow handling algorithm”**
- Pick the mid-key (say 90) in the node and move it to parent.
- Move everything to the right of 90 to the empty node c.
5. Redistribute Non-Leaf with Neighbor

- Delete 20
  - Underflow at \( a \)? Min 2 ptrs, currently 3. Done
6. Reduce Tree Depth

- Delete 20
  - Underflow! Merge d with e.
  - Move everything in the right node to the left
6. Reduce Tree Depth

• Delete 20
  • From the parent node, delete pointer and key to the deleted node
6. Reduce Tree Depth

- Delete 20
  - Merge \( b \) and \( c \)
    - Pull down the mid-key 50 in the parent node
    - Move everything in the right node to the left.
6. Reduce Tree Depth

• Delete 20
  • After merging $b$ and $c$, remove empty root node
  • Tree depth is decreased by one
6. Reduce Tree Depth

- Delete 20
Important Points

• Remember:
  • For leaf node merging, we delete the mid-key from the parent
  • For non-leaf node merging/redistribution, we pull down the mid-key from their parent.

• Exact algorithm: Figure 14.21
Where does $n$ come from?

• $n$ determined by
  • Size of a node
  • Size of search key
  • Size of an index pointer

• Q: 1024B node, 10B key, 8B ptr $\rightarrow n$?

\[
\begin{align*}
8n + 10(n-1) &\leq 1024 \\
8n + 10n - 10 &\leq 1024 \\
18n &\leq 1024 + 10 = 1034 \\
\frac{1034}{18} &\leq n \\
n &\leq 57.44
\end{align*}
\]
Range Search on B+tree

• SELECT *
  FROM Student
  WHERE sid > 60?