Index Structures: Part 2
B+-trees

B+-trees

Just as simple index structures, B+trees are designed to index the content of existing database relations/data files in DBMS.

A B+-tree is a balanced tree data structure defined as follows:

- Each node in a B-tree consists of $n$ key values and $n + 1$ pointers. Figure below shows the node structure for $n = 3$. For simplicity, we will write that a node $N$ is a pair $\langle \text{Keys}[0..n], \text{Pointers}[0..n + 1] \rangle$.

B+-tree Node, n=3

<table>
<thead>
<tr>
<th>Keys:</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- B+-trees have three types of nodes: root, internal and leaf.

- Leaf nodes are constructed as follows:
  - At least half of the key value slots in each leaf node is not empty.
  - Given a leaf node $l = \langle \text{Keys}[\cdot], \text{Pointers}[\cdot] \rangle$, if $l.\text{Keys}[i] = a$ (not empty) then $l.\text{Pointers}[i]$ contains a pointer to a record with key value $a$ in the data file.
  - $l.\text{Keys}[i] \leq l.\text{Keys}[i + 1]$ for all non-empty slots.
  - The last pointer of the leaf node, $l.\text{Pointers}[n + 1]$ points to the next leaf node $l'$. If $k$ is the number of non-empty key values in $l$, then $l.\text{Keys}[k] \leq l'.\text{Keys}[1]$. 
The structure of leaf nodes is illustrated below.

- **Internal nodes** have the following structure:
  - Each internal node has at least half of its key value slots occupied.
  - Given an internal node $N = (\text{Keys}, \text{Pointers})$, if $N.\text{Keys}[i] = a_i$ is non-empty, then $N.\text{Pointers}[i] \neq \text{NULL}$ and points to a node in the next level of the $B+$-tree. This node may be a leaf node, or another internal node.
  - If $N.\text{Pointers}[i] = N'$, then for $j \leq n$, if $N'.\text{Keys}[j]$ is not empty, then $N'.\text{Keys}[j] \leq N'.\text{Keys}[i]$.
    Additionally, if $i > 1$, $N'.\text{Keys}[j] \geq N.\text{Keys}[i - 1]$.
  - If $N.\text{Keys}[n] = a_n$ is not empty, then $N.\text{Pointers}[n + 1] \neq \text{NULL}$ points to a node $N'$ in the next level of the $B+$-tree, and all nonempty keys $N'.\text{Keys}[j] \geq N.\text{Keys}[n]$.

The structure of internal nodes is shown below:

- **Root node.** The structure of the root node of the $B+$-tree is similar to the structure of the internal node, with the exception that root node may contain
fewer than half of its key value slots occupied. Instead, at least 2 pointers (and 1 key) in the root node must be non-empty.

A simple $B+$-tree for $n = 3$ is shown below:

![B+-tree diagram]

**$B+$-trees and Indexing Database Records**

$B$-trees, and $B+$-trees are balanced trees with a guarantee that beyond the root node, all other nodes are rather dense (i.e., filled at 50% or more).

The search algorithm over $B$-trees and $B+$-trees is straightforward:

- given a key value $X$, starting at the root node, traverse the key values stored in the node, until a value $Y > X$ is discovered at some slot $i$.
- Retrieve the node $Pointers[i]$.
- If all non-empty key values in the node are smaller than $X$, follow the last non-empty pointer.

$B+$-trees are an adaptation of the standard $B$-tree structure to the secondary storage. Each node of a $B+$-tree has the size of one disk block. The data portion of the disk block is broken into $n (Key, Pointer)$ pairs, and an additional, $n + 1$st pointer is stored at the end of the page.

The second distinction of $B+$-trees is the fact that all leaf nodes are linked with each other. This makes it easy to search for keys in a sequence: searching for a starting position is done by traversing the tree, but after the first leaf node is
retrieved, one can follow the \( n + 1 \)st pointer on the page, to retrieve the next leaf node.

**Note:** we also note that while the standard structure of a \( B+ \)-tree assumes only a single-linked list of leaf nodes, we can also store a pointer to previous leaf node in the block header of each leaf node page.

\( B+ \)-trees can be used to store any of the index structures discussed before:

- **Dense indexes** on sequential files. The leaf nodes form the dense index, while the upper layers provide fast navigation to the necessary key.
- **Sparse indexes** on sequential files. Same as above, leaf nodes form the sparse index.
- **Secondary indexes.** Leaf nodes present all key occurrences in sorted order.
- **Indexes with duplicate keys.** \( B+ \)-trees need to be slightly updated to allow for seamless indexing of data with duplicate keys. In particular, the meaning of a key in an internal node has to change somewhat.

**How many layers?**

Suppose our disk blocks contain 4Kb each, 4096 bytes. Let our key values be integers, 4 bytes long and let our pointers be 8 bytes in size.

How many key values can we store in a single node?

We know that \( 12n + 8 + \text{HeaderSize} \leq 4096 \). If we take \( \text{HeaderSize} \) to be 80 bytes, this would lead to \( 12n = 4008 \), or

\[
    n = 334.
\]

A one-level \( B+ \)tree (root and leaves) can thus index \( 334^2 = 111,556 \) records. A two-level \( B+ \)tree (root, internal layer and leaves) can index \( 334^3 = 37,259,704 \) records.