B+Trees

Arnab Chakrabarti
Implementation of Databases
i5 RWTH Aachen
B+ Trees

- Each node stores up to $d$ references to children and up to $d - 1$ keys.
- Each reference is considered “between” two of the node's keys.
B+-Trees: Formal Definition

A B+-tree of type \((k, k^*)\) is a multi-path tree with the following properties:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node \(N\) with \(k\) being the number of keys in \(N\): all keys in the first child's subtree are less than \(N\)'s first key; and all keys in the \(i\)th child's subtree \((2 \leq i \leq k)\) are between the \((i-1)\)th key of \(N\) and the \(i\)th key of \(N\).
- The root has at least two children.
- Every non-leaf, non-root node has at least \(\lfloor d/2 \rfloor\) children.
- Each leaf contains at least \(\lfloor d/2 \rfloor\) keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.
B+ Tree: Not so formal definition

- B+ Tree: **Balanced tree** with intermediary nodes and leaf nodes
- Intermediary nodes contain only pointers / address to the leaf nodes. All leaf nodes will have records and all are at same distance from the root.
- Each intermediary node can have n/2 to n children. Only root node will have 2 children.
- Leaf node stores \((n-1)/2\) to \(n-1\) values
- As the number of intermediary nodes increases and hence the leaf nodes i.e. as B+ tree extends, the traversal speed increases log arithmetically \(\log_{(n/2)}(K)\)
- Records are in sorted order
- Since all the leaf nodes are at equal distance, the time for I/O fetch is much less. Hence the performance of the tree will also increase.
Why do we need B+ trees in databases?

Goals:
1. Sorted Intermediary and leaf nodes
2. Fast traversal and Quick Search

<table>
<thead>
<tr>
<th>STUDENT_ID</th>
<th>STUDENT_NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Joseph</td>
<td>Alaiedon Township</td>
</tr>
<tr>
<td>101</td>
<td>Allen</td>
<td>Fraser Township</td>
</tr>
<tr>
<td>102</td>
<td>Chris</td>
<td>Clinton Township</td>
</tr>
<tr>
<td>103</td>
<td>Patty</td>
<td>Troy</td>
</tr>
<tr>
<td>104</td>
<td>Jack</td>
<td>Fraser Township</td>
</tr>
<tr>
<td>105</td>
<td>Jessica</td>
<td>Clinton Township</td>
</tr>
<tr>
<td>106</td>
<td>James</td>
<td>Troy</td>
</tr>
<tr>
<td>107</td>
<td>Antony</td>
<td>Alaiedon Township</td>
</tr>
<tr>
<td>108</td>
<td>Jacob</td>
<td>Troy</td>
</tr>
</tbody>
</table>
Internal Data Structure of B+ Tree
B⁺-Tree: Search

- Search for value p
  - Start with root node
  - Search in every node from left to right
  - Inner node
    - Compare each value V with p and follow the corresponding branch
  - Leaf node
    - If p exists => success!
    - Otherwise: Value p does not exist in the tree

- Costs: h page accesses (search always stops at the leaf level)
  (h: height of the tree)
B+-Tree: Insert (1)

**Insertion algorithm**

- Descend to the leaf where the key fits.
- If the node has an **empty space**, insert the key/reference pair into the node.
- **Redistribute Phase**: If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes.
  - If the node is a leaf:
    - take a copy of the **minimum value in the second of these two nodes** and repeat this insertion algorithm to insert it into the parent node.
  - If the node is a non-leaf:
    - exclude the **middle value during the split** and repeat this insertion algorithm to insert this excluded value into the parent node.
**B⁺-Tree: Insert (2)**

- **Insert 5**
  - Current tree: 2, 4, 6, 7, 22, 24, 26, 90
  - New node: 5, 7
  - Updated tree: 2, 4, 5, 6, 7, 22, 24, 26, 90

- **Insert 25**
  - Current tree: 2, 4, 5, 6, 7, 22, 24, 25, 26, 90
  - New node: 25
  - Updated tree: 2, 4, 5, 6, 7, 22, 24, 25, 26, 90

- **Parameters**
  - \(k = 1\)
  - \(k^* = 2\)
More Examples: Insertion

Initial:

Insert 20:

Insert 13:

Insert 15:
More Examples: Insertion

Insert 10:

Insert 11:

Insert 12:
Deletion Algorithm

- Remove the required key and associated reference from the node.
- If the node still has enough keys and references to satisfy the invariants, stop.
- **Redistribute Step**: If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
- **Merge Step**: If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

➢ Always see that the node is half full or more (stable node)!
**B⁺-Baum: Löschen (2)**

**Steps:**
1. **Delete**
2. **Borrow**
3. **Repair**

![Diagram of B⁺-Tree with deletion steps](image)

- **Delete 7** ("Stealing")
- **k = 1**
- **k* = 2**
B⁺-Baum: Löschen (3)

Delete 5

(Step 1, "Merge")

Steps:
1. Delete
2. Merge
3. Delete Parent (iteratively)

(Step 2, B-Tree Merge)
B-Trees vs. B+-Trees

- Properties of B+-Tree compared to B-Tree:
  - Inner nodes only contain pointers
  - In B Tree: Each node will have only two branches and each node will have some records. Hence here no need to traverse till leaf node to get the data.
  - Partially, keys are stored redundantly
  - For each data record access h pages have to be read
  - Linkage of leaf nodes enables sequential iteration of data records in sorting order

- Operations supported in B+Trees:
  - Direct key access
  - Sorted sequential key access
  - Exact queries, prefix match queries, range queries, queries for extreme values
Block Size=512B

1000 Records