Spring 2020 – Online Instruction Plan

- Week 1 (March 30 – April 2):
  - File Organization and Overview of Indexes
  - B+-Trees
  - Hashing
  - Miscellaneous topics in Indexes

- Week 2: Query Processing

- Week 3: Transactions 1

- Week 4: Transactions 2

- Week 5: Parallel Database and MapReduce
B+-Trees

- Book Chapters
  - 11.3

- Key topics:
  - B+-Trees as a multi-level index, and basic properties
  - How to search in a B+-Tree?
  - How to update B+-Tree when a new tuple in inserted in the relation?
    - Key challenge: keeping the index “balanced” and all the pages “sufficiently full”
  - How to handle a delete from the underlying relation?
    - Same key challenge
11.3.2 Queries on B+-Trees

Let us consider how we process queries on a B+-tree. Suppose that we wish to find records with a search-key value of $V$. Figure 11.11 presents pseudocode for a function find() to carry out this task. Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest $i$ such that search-key value $K_i$ is greater than $V$. Then, if $V < K_i$, the function moves to the left child; otherwise, it moves to the right child. This process continues until a leaf node is reached or a null child is encountered.

These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B+-tree. Indeed, the "B" in B+-tree stands for "balanced." It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

Observe that the height of this tree is less than that of the previous tree, which had $n = 4$. These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B+-tree. Indeed, the "B" in B+-tree stands for "balanced." It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.
B⁺-Tree Node Structure

- Typical node

| $P_1$ | $K_1$ | $P_2$ | ... | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

- $K_i$ are the search-key values
- $P_i$ are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$
Properties of B+-Trees

- **It is balanced**
  - Every path from the root to a leaf is same length

- **Leaf nodes (at the bottom)**
  - $P_1$ contains the pointers to tuple(s) with key $K_1$
  - ...
  - $P_n$ is a pointer to the next leaf node
  - Must contain at least $n/2$ entries

| $P_1$ | $K_1$ | $P_2$ | ... | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
Properties

- **Interior nodes**

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$K_1$</th>
<th>$P_2$</th>
<th>$\ldots$</th>
<th>$P_{n-1}$</th>
<th>$K_{n-1}$</th>
<th>$P_n$</th>
</tr>
</thead>
</table>

- All tuples in the subtree pointed to by $P_1$, have search key $< K_1$
- To find a tuple with key $K_1' < K_1$, follow $P_1$
- $\ldots$
- Finally, search keys in the tuples contained in the subtree pointed to by $P_n$, are all larger than $K_{n-1}$
- Must contain at least $n/2$ entries (unless root)
B+-Trees - Searching

- How to search?
  - Follow the pointers

- Logarithmic
  - $\log_{B/2}(N)$, where $B = \text{Number of entries per block}$
  - $B$ is also called the order of the B+-Tree Index
    - Typically 100 or so

- If a relation contains 1,000,000,000 entries, takes only 4 random accesses

- The top levels are typically in memory
  - So only requires 1 or 2 random accesses per request
Example B+-Tree Index

Observe that the height of this tree is less than that of the previous tree, which had $n = 4$. These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B+-tree. Indeed, the “B” in B+-tree stands for “balanced.” It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

11.3.2 Queries on B+-Trees

Let us consider how we process queries on a B+-tree. Suppose that we wish to find records with a search-key value of $V$. Figure 11.11 presents pseudocode for a function find() to carry out this task. Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest $i$ such that search-key value $K_i$ is greater than the search-key value.
If this were a “primary” index, then not all ”keys” are present in the index.
**Tuple Insertion**

- Find the leaf node where the search key should go
- If already present
  - Insert record in the file. Update the bucket if necessary
    - This would be needed for secondary indexes
- If not present
  - Insert the record in the file
  - Adjust the index
    - Add a new \((K_i, P_i)\) pair to the leaf node
    - Recall the keys in the nodes are sorted
  - What if there is no space?
**Tuple Insertion**

- Splitting a node
  - Node has too many key-pointer pairs
    - Needs to store $n$, only has space for $n-1$
  - Split the node into two nodes
    - Put about half in each
  - Recursively go up the tree
    - May result in splitting all the way to the root
    - In fact, may end up adding a *level* to the tree
  - Pseudocode in the book!!
Let us consider how we process queries on a B\textsuperscript{+} tree. Indeed, the height of this tree is less than that of the previous tree, which had the same set of search-key values (the search-key values \( n \) and \( 2n \)) and a pointer to the root node.

Figure 11.10 presents the algorithm for lookup, which is to find the search-key value \( K \) in the existing node and the remaining values in a newly created node. The function \( find() \) must then be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

Figure 11.11 represents the pseudocode for insertion into the B\textsuperscript{+} tree for Figure 11.9. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on insertion of Brandt. We then need to insert an entry for Adams into the B\textsuperscript{+} tree of Figure 11.9. It was possible to perform this insertion without splitting a leaf node.

Using the algorithm for lookup, we find that the value of search-key being inserted is greater than the search-key value in the leaf node on the right (Figure 11.13). The new node has already been created (Figure 11.14). This new right-hand-side node contains the search-key value being inserted into the B\textsuperscript{+} tree and a pointer to the leaf node in which the value would be inserted. In the worst case, all nodes along the path to the root would have to be split, requiring an entire tree to become deeper.

Figure 11.15 shows the result of splitting a nonleaf node. The new right-hand-side node contains the search-key value being inserted into the B\textsuperscript{+} tree and a pointer to the leaf node in which the value would be inserted. In the worst case, all nodes along the path to the root would have to be split, requiring an entire tree to become deeper.

We now consider an example of insertion in which a node must be split. Assume that the search-key value \( K \) is to be added to the existing node and the remaining values in a newly created node. If there were no room, the parent would have had to be split, requiring an entire tree to become deeper.

Figure 11.13 shows the result of inserting a record with search key 1. It was possible to perform this insertion without splitting a leaf node.

**Figure 11.13** Insertion of “Adams” into the B\textsuperscript{+} -tree of Figure 11.9.
**B+-Trees: Insertion**

![B+-Tree Diagram](image)

**Figure 11.14** Insertion of “Lamport” into the B+-tree of Figure 11.13.
Updates on B⁺-Trees: Deletion

- Find the record, delete it.
- Remove the corresponding (search-key, pointer) pair from a leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care?
  - Solution:
    1. See if you can borrow some entries from a sibling
    2. If all the siblings are also just barely full, then merge (opposite of split)
- May end up merging all the way to the root
- In fact, may reduce the height of the tree by one
Examples of B⁺-Tree Deletion

Figure 11.16  Deletion of “Srinivasan” from the B⁺-tree of Figure 11.13.
Another B+Tree Insertion Example

INITIAL TREE

Next slides show the insertion of (125) into this tree
According to the Algorithm in Figure 12.13, Page 495
Another Example: INSERT (125)

Step 1: Split L to create L’

Insert the lowest value in L’ (130) upward into the parent P
Another Example: INSERT (125)

Step 2: Insert (130) into P by creating a temp node T
Another Example: INSERT (125)

Step 3: Create P’; distribute from T into P and P’

New P has only 1 key, but two pointers so it is OKAY. This follows the last 4 lines of Figure 12.13 (note that “n” = 4) K” = 130. Insert upward into the root
Another Example: INSERT (125)

Step 4: Insert (130) into the parent (R); create R'

Once again following the insert_in_parent() procedure, $K'' = 1000$
Another Example: INSERT (125)

Step 5: Create a new root
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: $133^3 = 2,352,637$ entries
  - Height 4: $133^4 = 312,900,700$ entries
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
B+ Trees: Summary

- **Searching:**
  - $\log_d(n)$ – Where $d$ is the order, and $n$ is the number of entries

- **Insertion:**
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching

- **Deletion**
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly