Lab 3: B+ Tree 1

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Outline

- Project Overview
- Assignment
- Appendix: Background on B+Tree Index

Project Overview

- The third programming project is to implement an **index** in your database system. The index is responsible for fast data retrieval without having to search through every row in a database table, providing the basis for both rapid random lookups and efficient access of ordered records.
- You will need to implement a B+Tree dynamic index structure. It is a balanced tree in which the internal pages direct the search and leaf pages contain actual data entries. Since the tree structure grows and shrinks dynamically, you are required to handle the logic of split and merge.
  - Over the next 3 labs, you will need to implement three Page classes to store the data of your B+Tree tree:
    - **B+Tree Parent Page** (Today's Assignment)
    - **B+Tree Internal Page** (Today's Assignment)
    - **B+Tree Leaf Page** (Next lab's Assignment)

Project Overview (Cont.)

- Like the last project, we are providing you with stub classes that contain the API that you need to implement. You should **not** modify the signatures for the pre-defined functions in these classes. If you do this, then it will break the test code that we will use to grade your assignment and you end up getting no credit for the project. If a class already contains certain member variables, you should **not** remove them. But you may add private helper functions/member variables to these classes in order to correctly realize the functionality.
- The correctness of B+Tree index depends on the correctness of your implementation of buffer pool, we will **not** provide solutions for the previous programming projects.
- This is a single-person project that will be completed individually (i.e. no groups).

Outline

- Project Overview
- Assignment
- Appendix: Background on B+Tree Index

Preparation to work on Lab 3

- We will provide you the solution of the following files on LMS as a zip file “lab3-prep.zip”.
  - src/include/storage/index/b_plus_tree.h
  - src/storage/index/b_plus_tree.cpp
  - src/include/storage/index/index_iterator.h
  - src/storage/index/index_iterator.cpp
- Download and use the solution provided and focus on implementing Page classes for the B+Tree index.

Assignment: B+ Tree Parent Page

B+Tree Parent Page

B+Tree Parent Page is in `src/include/storage/page/b_plus_tree_page.h` and `src/storage/page/b_plus_tree_page.cpp`

This is the parent class that both the Internal Page and Leaf Page inherited from and it only contains information that both child classes share. The Parent Page is divided into several fields as shown by the table below.

B+Tree Parent Page Content

Variable Name	Size	Description
page_type_	4	Page Type (internal or leaf)
lsn_	4	Log sequence number (Used in Project 4)
size_	4	Number of Key & Value pairs in page
max_size_	4	Max number of Key & Value pairs in page
parent_page_id_	4	Parent Page Id
page_id_	4	Self Page Id

You must implement your Parent Page in the designated files. You are only allowed to modify the header file (src/include/storage/page/b_plus_tree_page.h) and its corresponding source file (src/page/b_plus_tree_page.cpp).

Assignment: B+ Tree Internal Page

B+Tree Internal Page

B+Tree Internal Page is in `src/include/storage/page/b_plus_tree_internal_page.h` and `src/storage/page/b_plus_tree_internal_page.cpp`

Internal Page does not store any real data, but instead it stores an ordered **m** key entries and **m+1** child pointers (a.k.a page_id). Since the number of pointers does not equal the number of keys, the first key is set to be invalid, and lookup methods should always start with the second key. At any time, each internal page is at least half full. During deletion, two half-full pages can be joined to make a legal one or can be redistributed to avoid merging, while during insertion one full page can be split into two.

You must implement your Internal Page in the designated files. You are only allowed to modify the header file (`src/include/storage/page/b_plus_tree_internal_page.h`) and its corresponding source file (`src/storage/page/b_plus_tree_internal_page.cpp`).

Outline

- Project Overview
- Assignment
- Appendix: Background on B+Tree Index

Appendix: Background on B+Tree Index

B+Tree

A B+Tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertion, and deletions in O(log(n)). It is optimized for disk-oriented DBMSs that read/write large blocks of data.

Almost every modern DBMS that supports order-preserving indexes uses a B+Tree. There is a specific data structure called a B-Tree, but people also use the term to generally refer to a class of data structures. The primary difference between the original B-Tree and the B+Tree is that B-Trees store keys and values in all nodes, while B+ trees store values only in leaf nodes. Modern B+Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the B^{link}-Tree.

Formally, a B+Tree is an M-way search tree (where M represents the maximum number of children a node can have) with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root is at least half full (M/2 − 1 <= num of keys <= M − 1).
- Every inner node with k keys has k+1 non-null children.

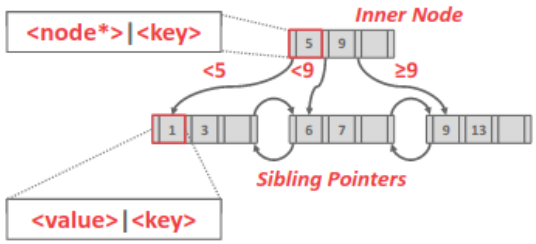

Figure 1: B+ Tree diagram

Appendix: Background on B+Tree Index (Cont.)

B+Tree (Cont.)

Every node in a B+Tree contains an array of key/value pairs. The keys in these pairs are derived from the attribute(s) that the index is based on. The values will differ based on whether a node is an inner node or a leaf node. For inner nodes, the value array will contain pointers to other nodes. Two approaches for leaf node values are record IDs and tuple data. Record IDs refer to a pointer to the location of the tuple. Leaf nodes that have tuple data store the actual contents of the tuple in each node.

Though it is not necessary according to the definition of the B+Tree, arrays at every node are almost always sorted by the keys.

Conceptually, the keys on the inner nodes can be thought of as guide posts. They guide the tree traversal but do not represent the keys (and hence their values) on the leaf nodes. What this means is that you could potentially have a key in an inner node (as a guide post) that is not found on the leaf nodes. Although it must be noted that conventionally inner nodes possess only those keys that are present in the leaf nodes.

Appendix: Background on B+Tree Index (Cont.)

B+Tree (Cont.)

Insertion

To insert a new entry into a B+Tree, one must traverse down the tree and use the inner nodes to figure out which leaf node to insert the key into.

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation is done.
   - Otherwise split L into two nodes L and L2. Redistribute entries evenly and copy up the middle key. Insert index entry pointing to L2 into the parent of L.
3. To split an inner node, redistribute entries evenly, but push up the middle key.

Deletion

Whereas in inserts we occasionally had to split leaves when the tree got too full, if a deletion causes a tree to be less than half-full, we must merge in order to rebalance the tree.

1. Find correct leaf L.
2. Remove the entry:
   - If L is at least half full, the operation is done.
   - Otherwise, you can try to redistribute, borrowing from siblings.
   - If redistribution fails, merge L and siblings.
3. If merge occurred, you must delete an entry in parent pointing to L.

Appendix: Background on B+Tree Index (Cont.)

B+Tree (Cont.)

Insertion

To insert a new entry into a B+Tree, one must traverse down the tree and use the inner nodes to figure out which leaf node to insert the key into.

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation is done.
   - Otherwise split L into two nodes L and L2. Redistribute entries evenly and copy up the middle key. Insert index entry pointing to L2 into the parent of L.
3. To split an inner node, redistribute entries evenly, but push up the middle key.

Deletion

Whereas in inserts we occasionally had to split leaves when the tree got too full, if a deletion causes a tree to be less than half-full, we must merge in order to rebalance the tree.

1. Find correct leaf L.
2. Remove the entry:
   - If L is at least half full, the operation is done.
   - Otherwise, you can try to redistribute, borrowing from siblings.
   - If redistribution fails, merge L and siblings.
3. If merge occurred, you must delete an entry in parent pointing to L.

Appendix: Background on B+Tree Index (Cont.)

B+Tree (Cont.)

Selection Conditions

Because B+Trees are in sorted order, lookups have fast traversal and also do not require the entire key. The DBMS can use a B+Tree index if the query provides any of the attributes of the search key. This differs from a hash index, which requires all attributes in the search key.

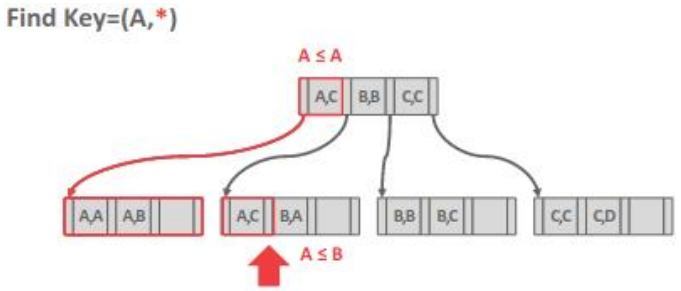

Figure 2: To perform a prefix search on a B+Tree, one looks at the first attribute on the key, follows the path down and performs a sequential scan across the leaves to find all they keys that one wants.

The End