**Report for Project 2: Implementing an Index Manager**

1. **Introduction and Assumptions:**
   1. Currently we only handle two types of index: integer and float.
   2. Assume key value will not duplicate, so we do not need to cope with overflow.
   3. The index is stored with fully functioned B+ tree structure, i.e., the tree will be balanced upon insertion and deletion.
2. **Design**
   1. **Data Structure on Disk**

Each index is stored as one file with the file name format:

**IX\_<table name>\_<attribute name>.idx**

This file is unique, which means trying to create an index for the same table same attribute will yield an error.

The first page (index 0) stores metadata of the index, sequentially including:

1. 4 bytes unsigned rootPageNum – the page number of root node in the file; the start point for initializing a tree in memory.
2. 4 bytes unsigned height – the height of index tree (starts from 1).
3. 4 bytes unsigned freePageNum – the page number of most recently released page due to deletion.

In order to reuse the pages after deletion of node, we maintain a stack of free pages, keeping the page number pointing to the top of this stack in the metadata. Each freed page will contain the page number of its prior page in the stack.

B+ tree index data start from the second page (index 1), with the following node formats:

1. Non-leaf node:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # of keys | Page # of child node | Key | Page # of child node | … | Key | Page # of child node |
| 4 bytes | 4 bytes | 4 bytes | 4 bytes |  | 4 bytes | 4 bytes |

1. Leaf node

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| # of keys | Page # of left entry | Page # of right entry | Key | RID | … … | Key | RID |
| 4 | 4 | 4 | 4 | 8 |  | 4 | 8 |

The order of B+ tree can be passed as a parameter while the tree is constructed. A default order is assigned if there is no explicit specification.

* 1. **Abstract Data Structure**

**B+ Tree Index (BTree<KEY>)**

We designed a relative independent **BTree<KEY>** class with template mechanism to represent B+ tree index in memory. The **struct BTreeNode<KEY>** represents each node in one tree, for both non-leaf node and leaf node. Please see implementation section for detailed structure.

The root node is read when the tree is created, whereas other nodes are lazy-load – being read upon necessity. For example, only the nodes on the path to locate one key are read during search function. However, the information of page numbers of nodes are read and stored in their parent’s childrenPageNum field, so nodes can be read if necessary.

* 1. **Play with Index**

When index manager creates an index, it first tries to read the catalog of given table to see whether or not the given attribute exists. If the attribute is not found, an error will be thrown out. If the attribute does exist, index manager creates an index file with the name depicted above, and writes the first page with metadata. When index manager opens an index handle, a file handle is assigned to the handle. The B+ tree index in memory is partially built only when it is necessary. When the index handle is closed, metadata are eventually written to index file.

* 1. **Other Noticeable Points**

Since function pointer does not support template, we implement Functor[[1]](#footnote-1) to fulfill this requirement. This is typically utilized to pass ReadNode function to BTree so that the tree structure is independent with the index file organization.

1. **Implementation**
   1. **BTreeNode<KEY>**

NodeType type // an enum indicating its node type (NON\_LEAF\_NODE or LEAF\_NODE)

BTreeNode<**KEY**>\* parent; // a pointer to its parent

BTreeNode<**KEY**>\* left; // a pointer to its closest left node

BTreeNode<**KEY**>\* right; // a pointer to its closest right node

**unsigned** pos; // its position in parent node (starts from 0)

vector<**KEY**> keys; // keys on this node

vector<RID> rids; // corresponding RIDs if LEAF\_NODE

vector<BTreeNode<**KEY**>\*> children; // pointers to corresponding children nodes if NON\_LEAF\_NODE

**int** pageNum; // its page number in the index file; -1 indicates unsaved page

**int** leftPageNum; // the page number of its closest left page; -1 means no left page – it is the most left one

**int** rightPageNum; // the page number of its closest right page; -1 means no right page – it is the most right one

vector<**int**> childrenPageNums; // a list of page numbers of its children nodes if NON\_LEAF\_NODE – this is for lazy load

* 1. **BTree<KEY>**

There are three major functions, searching, inserting and deleting entries respectively:

RC **SearchEntry**(**const** **KEY** key, BTreeNode<**KEY**> \*\*leafNode, **unsigned** &pos);

RC **InsertEntry**(**const** **KEY** key, **const** RID &rid);

RC **DeleteEntry**(**const** **KEY** key,**const** RID &rid);

During the operations (i.e., insert and delete entries) on the tree, nodes requiring updates (i.e., new nodes to be added and nodes with information changes) are recorded in one list \_updated\_nodes, while nodes to be deleted are stored in another list \_deleted\_nodes. These changes are flushed to index file at the end of the operations, avoiding duplicated writing operations for the same nodes. Then **ClearPendingNodes** function is invoked to clear these two buffer lists.

Several protected functions actually perform the essential operations on the tree:

1. **template** <**typename** **KEY**> RC **SearchNode**(BTreeNode<**KEY**> \*node, **const** **KEY** key, **const** **unsigned** height, BTreeNode<**KEY**> \*\*leafNode, **unsigned** &pos);

Recursively searches the given key value starting from the given node considering a given height, and sets the leaf node and its position value if found. If the given key is not found, a position for insertion is set for later use. A double pointer of leaf node is passed in so that the change on it can be leveraged outside the function

1. **template** <**typename** **KEY**> RC **Insert**(**const** **KEY** key, **const** RID &rid, BTreeNode<**KEY**> \*leafNode, **const** **unsigned** pos);

Inserts a KEY/RID pair to a leaf node at the given position, splitting this node if necessary and in turn invoking the function c). All fields in the **struct** BTreeNode are updated accordingly, and modified nodes are added to the list \_updated\_nodes.

1. **template** <**typename** **KEY**> RC **Insert**(BTreeNode<**KEY**> \*rightNode);

Recursively inserts one non-leaf node to its parent and split the parent node if necessary. A new root node is created if the number of key values in current root node exceeds limitation. All fields in the BTreeNode struct are updated accordingly, and modified nodes are added to the list \_updated\_nodes.

* 1. **Index Manager (IX\_Manager)**

Basically, this class performs operations literally as depicted on the project website.

* 1. **Index Handle (IX\_Handle)**

When index manager opens an index handle, a file handle is set and the index handle reads metadata from the index file. Before it is closed by invoking **Close** function, the index handle cannot be opened again; otherwise, an error will be thrown out. The **Close** function writes metadata into index file through **UpdateMetadata** function and resets the status of index handle. Since index handle is not implemented with template mechanism, it contains two index tree pointers for both integer and float, but only one of them is used at a time.

Besides, index handle is also responsible for reading and writing nodes:

1. **template** <**typename** **KEY**> BTreeNode<**KEY**>\* **ReadNode**(**const** **unsigned** pageNum, **const** NodeType nodeType);
2. RC **WriteNodes**(**const** vector<BTreeNode<**KEY**>\*> &nodes);

While an index tree is initialized, the **ReadNode** function is also passed to the tree as a function pointer.

* 1. **Index Scan (IX\_Scan)**

1. http://www.newty.de/fpt/functor.html#chapter4 [↑](#footnote-ref-1)