**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**

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
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 no explicit specification.

* 1. **Abstract Data Structure**

1. **B+ Tree (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. **Index Manager (IX\_Manager)**
2. **Index Handle (IX\_Handle)**
3. **Index Scan (IX\_Scan)**
   1. **Other noticeable points**

Since function pointer does not support template, we implement Functor[[1]](#footnote-1) to fulfill this requirement.

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>**

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