Report for Project 2: Implementing an Index Manager

I. Introduction and Assumptions:

1. Currently we only handle two types of indexes: 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.
4. IX_IndexHandle must be valid (i.e., it contains valid PF_FileHandle) when it is passed to IX_IndexScan for OpenScan; otherwise, an error code will be returned.

II. 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:

a) 4 bytes unsigned rootPageNum – the page number of root node in the file; the start point for initializing a tree in memory.
b) 4 bytes unsigned height – the height of index tree (starts from 1).
c) 4 bytes unsigned freePageNum – the page number of most recently released page due to deletion.
d) 4 bytes AttrType attrType – the type of the attribute being used to create current index.

In order to reuse the pages after deletion of nodes, 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:

a) 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

b) 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.

2. 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.

3. 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 index handle. The B+ tree index in memory is partially built only when it is necessary, e.g., during the search nodes are read from file when they are touched in the search path. When the index handle is closed, metadata are eventually written to index file.

4. Other Noticeable Points

Since function pointer does not support template, we implement Functor[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.

III.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)
```

[1] http://www.newty.de/fpt/functor.html#chapter4

```
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
```

2. 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:

a)
```
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

b)
```
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.

c)
```
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.

d)
```
template <typename KEY> RC BTree<KEY>::DeleteNLeafNode(BTreeNode<KEY>*
Node, unsigned nodeLevel, const KEY key, const RID &rid,int&
deletedChildPos);
```

Recursively delete in one non-leaf node. If deletedChildPos = -1, return with SUCCESS. Otherwise, it means that the child node in the position deletedChildPos was deleted; we have the following four cases:

(i) If the current node is the root of the tree, just delete the key and children pointer at position deletedChildPos. Insert current node into list _updated_nodes; If the current node becomes empty, we need to decrease the height of the tree by one with setting _height--. Set the child node of the current node as the new root of the tree. Insert the page number of the current node into list _deleted_pageNum.

(ii) If the current node is not the root and there are entries more than the order of the tree, delete the key and children pointer at position deletedChildPos. Set deletedChildPos = -1. Insert current old into list _updated_nodes.

(iii) If the current node is not the root and there are not enough entries and a sibling of the current node has extra entries to share, redistribute entries between the current node and the sibling node.
Update the key value in the parent node to be the minimum key in the right subtree. deletedChildPos = -1.Insert current node, sibling node and the parent node into list _updated_nodes.

(iv) If the current node is not the root and there are not enough entries and the sibling doesn't have extra entries to share, merge the current node and sibling node. Set deletedChildPos to be the position of the relative right node of the two nodes. Insert the left node and the parent node into list _updated_nodes. Insert the page number of the relative right node into list _deleted_pageNum.

e)
```
template <typename KEY> RC BTree<KEY>::DeleteLeafNode(BTreeNode<KEY>* Node,
const KEY key, const RID &rid,int& deletedChildPos);
```

Delete in one leaf node. Search the key in the leaf node, if not found, return an error code; if found, deleted the corresponding key and RID in the leaf node, then we also have the following four cases:

(i) If the current leaf node is the root of the tree, just insert current old into list _updated_nodes; If the node becomes empty, the tree becomes empty too and set _height = 0.

(ii) If the current leaf node is not the root and there are entries more than the order of the tree. Set deletedChildPos = -1. Insert current leaf node into list _updated_nodes.

(iii) If the current leaf node is not the root and there are not enough entries and a sibling of the current leaf node has extra entries to share, redistribute entries between the current node and the sibling node. Update the key value in the parent node to be the minimum key in the right leaf node. Set deletedChildPos = -1. Insert current node, sibling node and the parent node into list _updated_nodes,

(iv) If the current leaf node is not the root and there are not enough entries and the sibling doesn't have extra entries to share, merge the current leaf node and sibling node. Set deletedChildPos to be the position of the relative right node of the two nodes. Insert the left node and the parent node into list _updated_nodes. Insert the page number of the relative right node into list _deleted_pageNum.

3. Index Manager (IX_Manager)

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

4. Index Handle (IX_IndexHandle)

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:

```
a) template <typename KEY> BTreeNode<KEY>* ReadNode(const unsigned pageNum,
   const NodeType nodeType);
b) 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.

5. Index Scan (IX_IndexScan)

Opening an IndexScan sets its open status (field: `bool isOpen`) as true. Hence the same IndexScan cannot be open again until it is closed when its open status is set to false. Field `keyValue` stores the key value for all operations except for NO_OP and NE_OP, while field `skipValue` holds the key value for NE_OP.

GetNextTuple function reads next valid tuple if exists; otherwise, a code *END_OF_SCAN* is returned. If no entry in current index, *END_OF_SCAN* is also returned; otherwise, it processes the operation according to CompOp as following:

a) NO_OP and NE_OP:
Reads the left most value in the index and translates the operation into GE_OP. If `skipValue` is met for original NE_OP, ignore current RID and gets the next valid one.

b) GE_OP and LE_OP:
Translates these two operations into EQ_OP + GT_OP / LT_OP, respectively.

c) EQ_OP:
Leverages the search function of IndexHandle to get the exact match entry. If return code from the search function is *ENTRY_NOT_FOUND*, returns *END_OF_SCAN*.

d) GT_OP and LT_OP:
(i) Leverages the search function of IndexHandle to get the position of `keyValue` if exists; otherwise, the insertion position of `keyValue` will be available (the key value at this position is greater than `keyValue`, or the end position of one node).

(ii) If keyValue exists or keyValue does not exist but the position is the end of one node, tries to get nearby right or left entry, respectively; otherwise, sets RID for GT_OP or tries to get the nearby left entry for LT_OP. keyValue is in turn updated to new key just found.

(iii) Performs above processes until left or right entry is not available, respectively.

6. List of ReturnCode

We assigned disparate code sections for different purpose: general codes are less than 10, codes for IX_Manager are 10 to 19, IX_IndexHandle 20 to 29, and IX_IndexScan 30 to 39. Since varying layers return their own code, one error may lead to multiple outputs of error messages. Therefore we may readily trace error messages to the root cause. Codes are:

```
SUCCESS = 0,                // Success

// general codes
INVALID_OPERATION = 1,      // Invalid operation, such as opening index handle twice
FILE_OP_ERROR,              // Error during operating a file, such as writing page
EMPTY_TREE,                 // Indicates the tree is empty

// IX_Manager
ATTRIBUTE_NOT_FOUND = 11,   // Attribute in given table cannot be found
CREATE_INDEX_ERROR,         // Error when creating an index
DESTROY_INDEX_ERROR,        // Error when destroying an index
OPEN_INDEX_ERROR,           // Error when opening an index
CLOSE_INDEX_ERROR,          // Error when closing an index

// IX_IndexHandle
KEY_EXISTS = 20,            // Extra Credit
ENTRY_NOT_FOUND,            // Indicates the entry for given key does not exist
INSERT_ENTRY_ERROR,         // Error when inserting an entry
DELETE_ENTRY_ERROR,         // Error when deleting an entry
SEARCH_ENTRY_ERROR,         // Error when searching an entry
OPEN_INDEX_HANDLE_ERROR,    // Error when opening an index handle
CLOSE_INDEX_HANDLE_ERROR,   // Error when closing an index handle

// IX_IndexScan
END_OF_SCAN = 30,           // Indicates the end of a scan; not an error
INVALID_INDEX_HANDLE,       // Index handle is valid to open an index scan
INVALIDE_INPUT_DATA,        // Invalid input data for scan, such as NULL key for EQ_OP
OPEN_SCAN_ERROR,            // Error when opening an index scan
CLOSE_SCAN_ERROR,           // Error when closing an index scan
```