Summary of Project 3 / Group 16

In project 3 we implemented the idea of B+ tree, and we also implemented varchar indexes. In our implementation, we choose the tree order to be 120. A page refers to a node, and a node contains a directory of (key, page number) for intermediate nodes, (key, RID) for leaf nodes. Our leaf node has a forward pointer which points to a sibling leaf. We used the recursive algorithm described in the textbook to do the insertion, and used lazy delete to do the deletion.

Here is how we implemented these classes:

1. IX_Manager

(1) Member Variables:

string INDEXDIR: to allocate all index files into a directory

PF_Manager* thisPFM: a pointer pointing to the PF_Manager singleton class

RM* thisRM: a pointer pointing to the RecordManager singleton class

These three variables will be initialized in IX_Manager's constructor.

(2) RC **CreateIndex**(const string tableName, const string attributeName)

The CreateIndex function first creates an empty file using PF_Manager*. Then it scans the whole data file via Record Manager pointer and finds all condition attribute values and the corresponding rids. Next, the IX_Manager opens this new created index with an IndexHandle. Initialize the root node of this index. And lastly, insert the scanned entries (key, rid) one by one until the B+ tree is completed.

(3) RC **DestroyIndex**(const string tableName, const string attributeName)

The DestroyIndex function firstly checks if this index file exists or not. If not, an error will be returned. If the index file exists, the IX_Manager will call PageFileManager to destroy this index file.

(4) RC **OpenIndex**(const string tableName, const string attributeName, IX_IndexHandle &indexHandle)

The OpenIndex function, as well, will first check if the index exists or not. Then it will configure the indexHandle to open the corresponding index file. Also the function will tell what AttrType of this index is to the indexHandle for further operation.

(5) RC **CloseIndex**(IX_IndexHandle &indexHandle)

CloseIndex will terminate the connection between an indexHandle and its connected index file. After closed, the indexHandle can no longer operate on an index file until opening an index again.

2. IX_IndexHandle

(1) RC **InsertEntry**(void *key, const RID &rid)

To insert an entry, the InsertEntry use a recursive function called

RC InsertEntryToNode(PageNum current, void *key, const RID &rid, void *newkey, PageNum &newpage)

The basic logic of InsertEntryToNode is that it starts from the root node, looing down the B+ tree to find the correct position in a leaf for insertion. There are two conditions here. If this leaf has free space, that is fine. If this leaf does not have enough space, this leaf has to split into two halves, left and right. And a new child entry must be return up to intermediate node. The new child entry consists of two parts. The first part is the leftmost key value in the right half. The second part is the page number of this right half node.

When recursively goes back up, we need to check if new child entry is empty. If so, that's fine, just keep recursively going up. But if new child entry is not empty, we need to insert this new child entry into current node as well. If there is enough space, we will insert new child entry and make new child entry empty. However, the similar situation may occur again if the current node does not have enough space for this new child entry. This node has to be split again. One special situation is that if the root node has to be split, we need to arrange a new root node by using the new child entry.

(2) RC **DeleteEntry**(void *key, const RID &rid)

To do the deletion we used a function called

RC DeleteEntryFromNode(PageNum parent, PageNum current, PageNum leftnode, void *key, const RID &rid, void *key_to_remove, PageNum &page_to_remove)

In the beginning of a deletion, we first find the corresponding leaf page that has the given (key, RID) using the same logic in the InsertEntry function. If it is not found, we'll return -1 to indicate that there's no such tuple in the tree. If it is found, we'll delete the entry in the leaf. After that, we'll check if a leaf is empty or not. If it is empty, than we will set the page_to_remove variable to notify its parent to delete the entry that points to the leaf. If a

intermediate node becomes empty, we'll recursively delete the corresponding entry in it parent.

3. IX_IndexScan

(1) RC **OpenScan**(const IX_IndexHandle &indexHandle, CompOp compOp, void *value)

We use two kinds of methods to implement the scan. When the comparison operator is EQ_OP, GT_OP or GE_OP, we use the same logic in the InsertEntry function to search for the leaf corresponding to the given key value, and start scanning leafs using forward pointers to find all qualified record ids. On the other hand, when the comparison operator is NE_OP, LT_OP or LE_OP, we start from the leftmost leaf to scan all leafs that matches the query condition. For NO_OP operator, we simply scan through all leafs to find all record ids. Once we find a record id whose key matches the query condition, we remember it in a queue for later use.

(2) RC **GetNextEntry**(RID &rid)

After we get all record ids that match the query condition, every time GetNextEntry is called, it will pop out a value from the queue and return the rid. If the queue is empty, we'll return IX_EOF to indicate there are no rids.

(3) RC **CloseScan**()

In the function we simply delete the memory occupied by the queue.

4. Extra credits

We implemented the support for varchar indexes. Compared to fixed key value like integer and float, strings don't have a fixed length so there's no constant order d. As a result, we will split the node if the it cannot contain the string, and the number of keys we store in a node depend on the size of keys in it.