Project 2 Report / Group ID: 16

Basic Ideas and Catalog Implementation

Our record manager stores a table with two files, one is a paged file that stores records, another one is a bitmap file that indicates whether a page is full or not. (These files are placed under a directory called "Tables".) We also have a catalog for storing table information. How we implement the catalog is by the implementation of our table structure, that is, the catalog itself is also treated as a table and we store it using the record manager. The attributes of the catalog are table name (TableName), attribute name (ColName), attribute type (Type), attribute length (Length), schema number (Schema), and page file name (PageFileName). The Schema attribute was originally set for the extra credit (add and delete attribute), but in the end we didn't have time to do these two functions so we left the schema number there. We have also implemented the reorganizeTable function.

Record Representation

In our record representation, we use an array of offsets to indicate the fields in the record. We also append a 2-byte schema number at the beginning of each record. The schema number was set for (a) the extra credit, but we didn't have time to finish it (b) indicating if the tuple is a pointer that points to another record in the paged file.

Page format

In a single page we use the first two bytes to represent the current offset that points to the remaining free space, and the second two bytes to represent the number of slots in it. For slots, we store them from the end of the page, each occupying 4 bytes: two for the offset of the record, two for the length of the record. Remaining spaces in the page are used for storing records.

Classes and Functions

Create and delete the record manager (RM constructor and destructor)
In the constructor, most of the work is to initialize the catalog. First, we open the catalog, if the catalog is not existed we'll create a new one. Then we'll keep a file handle for the catalog, so later we can insert, update, or delete table information using this file handle. Next we hard code the attributes of the catalog and store it as a class variable, and the attributes are used when we want to access the catalog table. Since in our implementation, the catalog is also a table constructed by the record manager, but the catalog does not have a catalog to describe its attributes, so we hard code and remember it as a class variable and we can use it if we want to access the catalog later in some functions. In the destructor what we do is just closing the file handle of the catalog.

createTable

In this function we first scan the catalog to check whether the same table name exists or not. If it doesn't exist, we open a paged file and a bitmap file for that table, then insert the table information to the catalog based on the given attributes.

deleteTable

In this function we first scan the catalog to check whether the same table name exists or not. If it exists, we delete its paged file and bitmap file, then remove its information in the catalog.

insertTupleToFile, deleteTupleFromFile, readTupleFromFile, scanFromFile

Before we introduce other functions, we have to mention about some functions we created in this project, especially these four functions. What these functions do is basically a part of a function. For example, in the insertTuple function, we'll call the insertTupleToFile function, which takes the same arguments as insertTuple except the table name is changed to a file handle. Why we do this is because we want to directly access the file handle of the catalog. In our implementation, since our catalog is created as a table using the record manager, we might want to use the insertTuple function to insert a table's information to the catalog. However, we can't call insertTuple directly since it will first check whether the table is in the catalog or not, so if we use the insertTuple function to insert, it will become a endless cycle: when we want to insert something to the catalog, it will first check the catalog about the catalog's information, which doesn't make sense. We solve this problem by hard coding the attributes of the catalog so we don't have to check some files to get the catalog's attributes before we want to insert something to it. And we also keep a file handle of the paged file of the catalog so it is treated as a separate table and we don't have to reserve a name for the catalog. That is, if we want to insert something to the catalog, instead of calling insertTuple, we call insertTupleToFile with the file handle of the catalog. The other functions deleteTupleFromFile, readTupleFromFile and scanFromFile are based on the same idea.

getAttributes

We use the scanFromFile function to scan the catalog to get a table's attributes. After we get the attribute data, we parse and store it in the given vector.

insertTuple

First we use the getAttributes function to get the table's attribute (we'll check whether the table exists or not), then we call the insertTupleToFile function to insert the record to the corresponding paged file. In our insertion, we first parse the record to get its length, and we check the bitmap file to get free page. Once we find a free page we'll check whether the remaining free space is enough for the record, if not enough we'll keep finding until we find a page that has enough space to allocate the record. If we can't find any free page in the paged file, we'll append a new page to allocate the record. We used the function formatData to parse the original data representation to our representation. After allocating the record, we'll check the remaining free space. In our implementation, if the remaining free space is less then 20 bytes, we'll mark this page as full in the bitmap, so the next time we insert a record we'll know that this page is full and is unavailable for allocating new records.

deleteTuple

First we use the getAttributes function to get the table's attribute (we'll check whether the table exists or not), then we'll call the deleteTupleFromFile function to delete the record in the paged file. After we find the record with the given rid, we check its first two bytes to make sure whether it is a pointer or not. If it is not a pointer, we know it's really a record, so we delete it by setting its offset in the slot to be -1. If it is a pointer, we use the pointer's information to get the record id of the updated tuple and recursively delete the tuple with this rid.

deleteTuples

In this function we first check whether the table exists or not. If it exists, we delete the paged file and bitmap file of the table, and create a new empty paged file and bitmap file for it.

readTuple

In this function we first check whether the table exists or not. If it exists, we call the readTupleFromFile with the corresponding file handle of its paged file. Then we'll use the given record id to find the record. After we find the record, we check its first two bytes to make sure whether it is a pointer or not. If it is not a pointer, we know it's really a record, so we parse the record from the paged file and copy it to the given buffer and return. If it is a pointer, we use the pointer's information to get the record id of the updated tuple and recursively read the tuple with this rid.

readAttribute

In this function we first call readTuple to get the corresponding record, and call getAttributes to get the table's attributes. After we have the record and the attributes, we parse the record to extract the attribute "attributeName" from it by the getIthAttr function, and copy the data to the given buffer.

updateTuple

When updating a tuple, firstly figure out the length of current tuple with the record id and the length of new tuple (which is data, by the way). There are three situations that need to be handled separately.

(1) If current tuple is not pointing to another tuple && the length of current tuple is longer than that of new tuple. We just need to replace the old tuple with the new one and update the slot information.

(2) If current tuple is not pointing to another tuple && the length of current tuple is shorter than that of new tuple. Then we have to insert the new tuple somewhere else, get the new tuple's rid. And make the current tuple point to the new tuple.

(3) If current tuple is pointing to another tuple, then we need to delete the tuple being pointed to. And insert the new tuple, make current tuple point to this new tuple.

reorganizePage

When reorganizing a page, we need to read the original page into a buffer, and at the same time create an updated page to write new data. The reorganizing process is, for every slot in the original page, check if its offset equals -1 (which means this slot is deleted). If so, copy only the slot info to updated page, ignoring the tuple info. If this slot is not deleted, then copy both its slot info and tuple info into updated page. Since we

have only changed page offset (tuples may be allocated somewhere else within a page), their rids remain the same.
After all slots have been processed, write back updated page into file. Additionally, check if the updated page is full or free. Make corresponding correction in the bitmap file.

scan
In this function we first check whether the table exist or not. If it exists, we scan the corresponding paged file and compared the record with the given condition. If the record matches with the condition, we store filter the record and store it into the iterator (the iterator will be described next). How we filter the record to output the given attributes is by the getOriginalFormat function, in which we read the record and extract just the given attributes from it.

RM_ScanIterator
We implement this class with three vectors. One is for storing record id, one is for storing data, and one is for storing data length. We also have a current_tuple that is initialized to zero in the beginning of a scan. Each time when getNextTuple is called, we set the rid with the one we stored and copy our stored data to the given buffer. Why we store the length is for passing the length while we use the memcpy function. After each getNextTuple we'll increment the value of current_tuple by one, so if the value is the same as the size of our vector, we know we have iterated all matched records and we should return a RM_EOF.

Extra credit

reorganizeTable
The bigger picture of reorganizeTable is create a new table, copy all valid data from the original table. After processing, delete the original table. And rename the new table as the original one.
The detailed process is, read every tuple in the original table one by one. We insert valid original tuples into the new table, which guarantees that the slots in the new table are continuous. If a tuple is deleted (offset == -1), skip both this slot and the record – This is different from reorganizePage, which still copy the slot info even if the tuple is deleted. If a tuple is pointing to other tuple, then copy the info of pointed tuple, store the data into current new table slot. Then delete the pointed tuple in case we read it again and store it twice. If a tuple is neither deleted nor a pointer, just copy its record info and insert it into the new table.