**Project2 Report for CS222**

The contents and implemental details of this project are described step by step in the related functions below:

**RM constructor:**

We use two tables to record information of all the other tables: *systemCatalog* and columnInfo. In this function, we create these two files but write nothing in it.

Instead of using paged file, records in these files are directly written into them. The record format we design for *systemCatalog* is:

**struct** *tableInfo* {

**int** tId;

**char** tableName[128];

**char** fileName[128];

};

and for *columnInfo*:

**struct** columnInfo{

**int** tId;

**char** columnName[128];

AttrType columnType;

AttrLength columnLength;

};

**RM::createTable:**

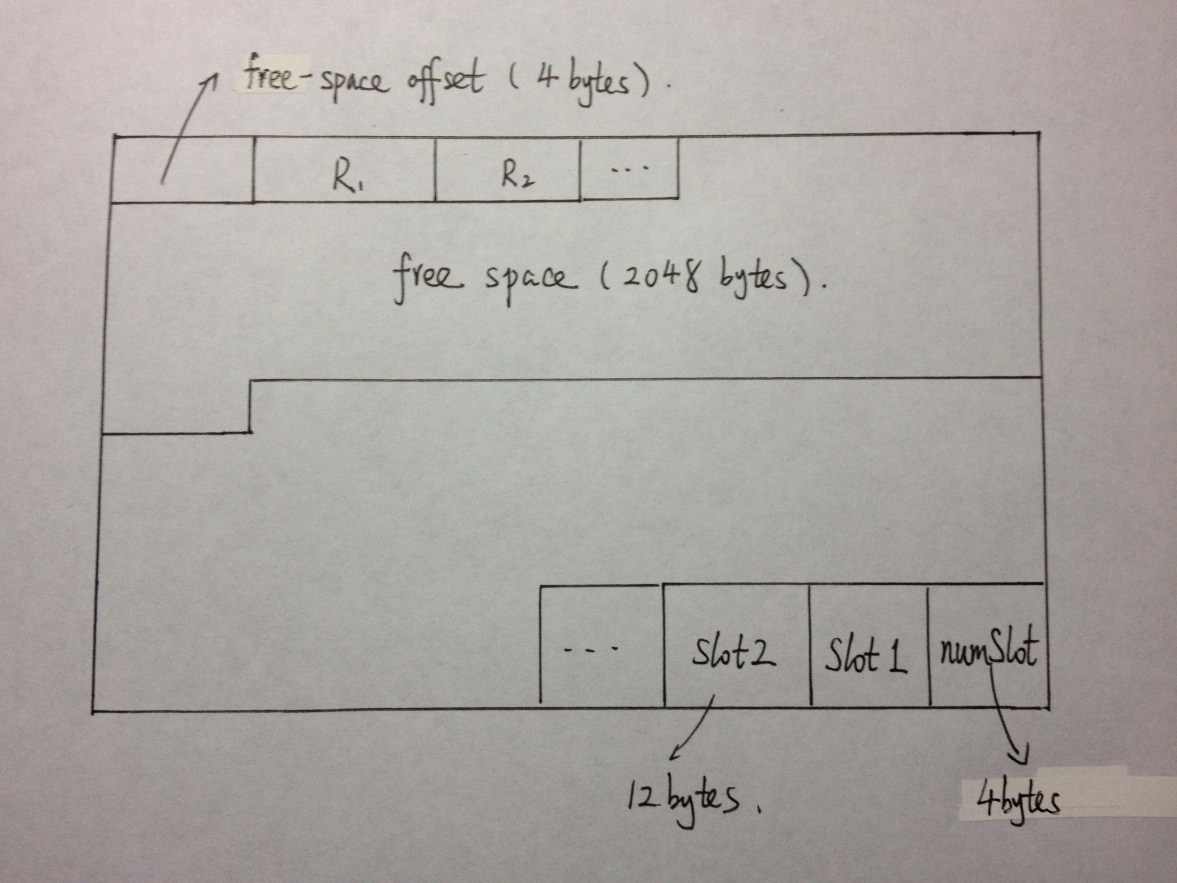
Several things are done in this function:

1. Update *systemCatalog* and *columnInfo*. a new *tID* is assigned to this table by searching the largest *tID* in *systemCatalog* and adding it by one. *fileName* is the same with *tableName*. Other items are written to the file according to the input arguments and given format.
2. Create a paged file named in *fileName*.
3. Initiate the directory page. We use page 0 in the file to record the free space of each following pages. We leave 2048 bytes for each page, so here in this function we write 2048 for the first 4 bytes, indicating page 1 is empty. And we then initiate other 4 bytes in page 0 to be -1, indicating they are not appended to this file yet.
4. Initiate page 1. The format of pages from page1 to the last page of this file is designed as shown in the picture below. Here in this function, we append page1 to the file, and initiate both free-space offset and numSlot to be 0.

**RM::deleteTable:**

Several things are done in this function:

1. Delete the paged file used to record this table.
2. Find the corresponding record in *systemCatalog* using *tableName*, and set its *tID* to be -1, indicating this table has been deleted.
3. Using the original *tID* to find the table’s columns in *columnInfo*, and set *tID* to be -1.



**RM::getAttributes:**

In this function, we do the following things sequentially:

1. Open the file *systemCatalog*. Read its contents *tableInfo* by *tableInfo*, and compare its *tableName* with the input argument, until it finds the equivalent one with *tID*.
2. If we can’t find a table with that name, or tID == -1, return -1. Otherwise, open the file *columnInfo*, and search for all the *columnInfo* with *tID*, return them in a vector of attribute and a success RC.

**RM::insertTuple:**

In this function, we do the following things sequentially:

1. Check the left free space in the specific page. This is done by reading out the corresponding four bytes in page0.
2. Calculate the length of the given tuple. This is done by referring to the *columnInfo*, if first attribute’s type is not *TypeVarChar*, add 4 to the total length. Otherwise, we should also add the length of the varchar in bytes, according to the number we read in the first 4 bytes. Repeat until all the attributes for *tableName* are considered.
3. Find the first qualified page which has enough free space to insert. If there is one in the already-existed page, read it out, find its free-space offset, and write the new tuple into it. Then create a new slot for it. The format of slot we use is:

**struct** slot{

**int** slotNum;

**int** offset;

**int** length;

};

The slotNum is gotten by reading the numSlotth slot’sslotNum and add it by one. Offset is the beginning of this tuple, and length is just what we compute in 2). At last, update the corresponding page0 place by minus length.

1. If there isn’t a qualified page, then we should append a new page to store it. The procedure is similar to 3) except doing some initializing thing.
2. Write the page from memory back to file. This step will be omitted in this document below.

**RM::deleteTuples:**

This can be done easily with the functions above :

1. *getAttributes* using tableName.
2. Delete the file filename.
3. *createTable* using the attributes we just got.

**RM::deleteTuple:**

1. Find the slot related with this tuple using RID.
2. Change the slotNum to -1.
3. Write this page back to file.

**RM::updateTuple:**

1. Check the length of the new tuple and the free space left for this page, just as in *insertTuple*.
2. If there is enough free space left, write the new tuple in the free space, change the Offset of slot and free space, and update the length of free space in page0.
3. If not, append a new page. Change the original *offset* of the slot to be -1, and the *length* to be the page number of the newly appended page. Then initialize the new page just like *insertTuple* and write the tuple into it.

**RM::readTuple:**

1. Find the corresponding page and slot according to RID.
2. If *slotNum* is -1, then the tuple has been deleted, return -1.
3. If *offset* is -1, then the tuple has been updated to a new page with the page number in *length*. We should read that page and locate at the first slot in it then.
4. Copy the tuple using memcpy, in *length* bytes.

**RM::readAttribute:**

1. *readTuple*
2. *getAttributes*
3. For each attribute, if its name does not equal to the input argument, we should move algong (four bytes for real of int, and additional length if it is a string).
4. If does, copy the memory. Four bytes for int or real. L+1 bytes for string (1 stands for the ‘/0’ indicating the end of string when we use it in the future without knowing its length).

**RM::reorganizePage:**

1. *Check if this page is a page which stores update record*
2. *If yes, return -1. Because there is always enough space in the update page*
3. *If not, initiate the new memory to store updated information*
4. *Iterate all the slots in the old page, copy every exist slot from old page to memory*
5. *Update offset in the slot to new position and copy tuple to new memory*
6. *Set the address of new free space position in memory*
7. *Update the length of free space in directory page*
8. *Write the memory block which store new information to page*

**RM::scan:**

1. *Pass all the parameters to scanIterator and all the operations are done in Iterator.*

**RM\_ScanIterator constructor:**

Pass variables: tableName, conditionAttribute, compOp, attributeNames to this constructor.

1. *Iterate all the tuples in the table*
2. *Check if condition is satisfied*
3. *Get all the valid tuples*
4. *Check what type of attributeName required*
5. *If int or real, just add the value to data*
6. *If varchar, first add the 4byte to store the length of varchar and add the varchar to data based on the length of it*

**RM::reorganizeTable:**

1. *Initiate a memory of one page size to store new information*
2. *Scan all the slots and store all the exist records*
3. *Copy tuple to new position and assign new rId to it*
4. *Update the offset of slot*
5. *Slot ++ until finishing scan*
6. *Update systemCatalog and columnInfo*