**CMPS 181 Project 2 Introduction**

In this project, you will continue implementing the record-based file manager (RBFM). Once you have finished implementing that, you will build a relation manager (RM) on top of the basic paged file system. The RM manager should meet the following basic requirements.

**Basic Requirements (100 points)**

**Catalog**

Create a catalog to hold all information about your database. This includes at least the following:

* Table information (e.g., table-name, table-id, etc.).
* For each table, the columns, and for each of these columns: the column name, type and length
* The name of the record-based file in which the data corresponding to each table is stored.

It is mandatory to store the catalog information by using the RBF layer functions. In other words, you should manually create the catalog's tables and populate them the first time your database is initialized. Once the catalog's tables (such as the **Tables** table with the following attributes (table-id, table-name, file-name) and the **Columns** table with the following attributes (table-id, column-name, column-type, column-length)) have been created, they should be persisted to disk. You need to make sure that you don't create the catalog tables unless it is the first time you start your program. All other subsequent invocations of your database should use the already created catalog's tables. Please note that users should be able to query your catalog's tables as any other table, but they should not be allowed to modify its content through the RM API. The modifications of the catalog's tables should be only allowed from system calls (for instance, when creating a user table or deleting a user table, etc.). To do this, you may want to have a "system" vs. "user" flag in your table RBF file so you can distinguish "system tables" from "user tables" and make it illegal to do anything but reading of "system" tables through the RM layer.

**Functions**

You need to finish the implementation of the record-based file manager (RBFM) that you started in part 1. Specifically, you should finish implementing the following methods:

**RecordBasedFileManager::deleteRecords()**, **RecordBasedFileManager::deleteRecord()**, **RecordBasedFileManager::updateRecord()**, **RecordBasedFileManager::readAttribute()**, **RecordBasedFileManager::reorganizePage()**, **RecordBasedFileManager::scan()**.

Please refer to the Project 1 Description for their explanations and look at the file **rbfm.h** for the signature of those methods. Once you finish the implementation of those methods and test them well, you will use those methods to implement the relation manager (please have a look at the file **rm.h**).

**RelationManager Class**

The RelationManager class is responsible of managing the database tables. It handles the creation and deletion of tables. It also handles the basic operations performed on top of a table (e.g., insert and delete tuples). Your program should create exactly one instance of this class, and all requests for RM component should be directed to that instance. The public methods of the class declaration are shown first, followed by descriptions of the methods. The last two methods in the class declaration are the constructor and destructor methods and are not explained further. **Note:** your tuple-oriented file system must create a relation manager (RM) that initializes catalog information you may need to store. It also creates a record-based file manager using the implementation in Project 1 and project 2.

class RelationManager

{

public:

RC createTable(const string &tableName, const vector<Attribute> &attrs);

RC deleteTable(const string &tableName);

RC getAttributes(const string &tableName, vector<Attribute> &attrs);

RC insertTuple(const string &tableName, const void \*data, RID &rid);

RC deleteTuples(const string &tableName);

RC deleteTuple(const string &tableName, const RID &rid);

// Assume the rid does not change after update

RC updateTuple(const string &tableName, const void \*data, const RID &rid);

RC readTuple(const string &tableName, const RID &rid, void \*data);

RC readAttribute(const string &tableName, const RID &rid, const string &attributeName, void \*data);

RC reorganizePage(const string &tableName, const unsigned pageNumber);

// scan returns an iterator to allow the caller to go through the results one by one.

RC scan(const string &tableName,

const string &conditionAttribute,

const CompOp compOp, // comparision type such as "<" and "="

const void \*value, // used in the comparison

const vector<string> &attributeNames, // a list of projected attributes

RM\_ScanIterator &rm\_ScanIterator);

protected:

RelationManager();

~RelationManager();

};

**RC createTable(const string &tableName, const vector<Attribute> &attrs)**

This method creates a table called tableName with a vector of attributes (attrs).

**RC deleteTable(const string &tableName);**

This method deletes a table called tableName.

**RC getAttributes(const string &tableName, vector<Attribute> &attrs);**

This method gets the attributes (attrs) of a table called tableName.

**RC insertTuple(const string &tableName, const void \*data, RID &rid);**

This method inserts a tuple into a table called tableName. You can assume that the input is always correct and free of error. That is, you do not need to check if the input tuple has the right number of attributes and if the attribute types match.

**RC deleteTuples(const string &tableName);**

This method deletes all tuples in a table called tableName. This command should result in an empty table.

**RC deleteTuple(const string &tableName, const RID &rid);**

This method deletes a tuple with a given rid.

**RC updateTuple(const string &tableName, const void \*data, const RID &rid);**

This method updates a tuple identified by a given rid. **Note:** if the tuple grows (i.e., the size of tuple increases) and there is no space in the page to store the tuple (after the update), then, the tuple is migrated to a new page with enough free space. Since you will implement an index structure (e.g., B-tree) in project 3, you can assume that tuples are identified by their rids and when they migrate, they leave a tombstone behind pointing to the new location of the tuple.

**RC readTuple(const string &tableName, const RID &rid, void \*data);**

This method reads a tuple identified by a given rid.

**RC readAttribute(const string &tableName, const RID &rid, const string &attributeName, void \*data);**

This method reads a specific attribute of a tuple identified by a given rid.

**RC reorganizePage(const string &tableName, const unsigned pageNumber);**

This method reorganizes the tuples in a page. That is, it pushes the free space towards the end of the page. **Note:** In this method you are NOT allowed to change the rids, since they might be used by other external index structures, and it's too expensive to modify those structures for each such a function call. It's OK to keep those deleted tuples and their slots.

**RC scan(const string &tableName, const string &conditionAttribute, const CompOp compOp, const void \*value, const vector<string> &attributeNames, RM\_ScanIterator &rm\_ScanIterator);**

This method scans a table called tableName. That is, it sequentially reads all the entries in the table. This method returns an iterator called rm\_ScanIterator to allow the caller to go through the records in the table one by one. A scan has a filter condition associated with it, e.g., it consists of a list of attributes to project out as well as a predicate on an attribute (“Sal > 40000”). **Note:** the RBFM\_ScanIterator should not cache the entire scan result in memory. In fact, you need to be looking at one (or a few) page(s) of data at a time, ever. In this project, let the OS do the memory-management work for you.

**RM\_ScanIterator Class**

The RM\_ScanIterator class is a class that represents an iterator which is used to go through the tuples in the table one by one. The way to use this iterator is as follow:

RM\_ScanIterator rmsi; rc = rm->scan(tableName, conditionAttribute, compOp, value, attributes, rmsi); while(rmsi.getNextTuple(rid, returnedData) != RM\_EOF){ //process the data; } rmsi.close();

The public methods of this class are shown next. The first two methods in the class declaration are the constructor and destructor methods and are not explained further.

class RM\_ScanIterator { public: RM\_ScanIterator(); ~RM\_ScanIterator(); // "data" follows the same format as RelationManager::insertTuple() RC getNextTuple(RID &rid, void \*data); RC close(); };

**RC getNextTuple(RID &rid, void \*data);**

This method is used to get the next tuple from the scanned table. It returns RM\_EOF when all tuples are scanned.

**RC close();**

This method is used to close the iterator.

**Design Assumptions**

You can make the following simplifying assumptions when implementing PFM, RBFM, and RM (including ScanIterator):

1. The size of one tuple cannot exceed the size of a file page. That is, an empty page can always hold at least one tuple. However, if a page has two (or more) tuples, then one of the tuples can grow to not fit on that page.
2. A table maps to a single file, and a single file contains only one table.

**Advanced Requirements (10 points)**

Advanced features are not required in your submission, and will be treated as extra credit work. Extra credit points will be tracked separately and used when considering effort as a factor when assigning final grades. Following are the advanced features:

public:

RC addAttribute(const string &tableName, const Attribute &attr);

RC dropAttribute(const string &tableName, const string &attributeName);

RC reorganizeTable(const string &tableName);

**RC addAttribute(const string &tableName, const Attribute &attr);**

This method adds a new attribute (attr) to a table called tableName. **Note:** This operation will update the catalogs but should not involve touching the data itself.

**RC dropAttribute(const string &tableName, const string &attributeName);**

This method drops an attribute called attributeName from a table called tableName. **Note:** This operation will update the catalogs but should not involve touching the data itself.

**HINT:** The above two methods will affect the way the operations that access the fields of the records work when accessing a tuple that was created before such a schema change.

**RC reorganizeTable(const string &tableName);**

This method reorganizes a table that causes reorganization of the tuples such that the tuples are collected towards the beginning of the file. Also, tuple redirection is eliminated. **Note:** In this case, and only this case, you are allowed to change the records rids since they might need to move to other pages.

**Explanation**

The commands listed above are by no means complete, but they do capture the essence of the tuple-oriented file system.

You have a lot of freedom in designing your specific algorithms and building your system. You should spend a significant amount of time in coming up with a design of your system before you start coding. A principal challenge will be the design of the catalog storage that stores information about tables.

We advise you to use memcpy to copy the int/real/char\* values into the tuple byte array and vice-versa. So, as long as the same machine does the reading and writing, things should be fine.

Grading will be based on the correctness of the implementation.