**Project 2 for CMPS 181: Implementing a Relation Manager**

**Note: Students should work in teams of two on project 2.**

**Introduction**

In project 2, you will continue implementing the record-based file manager (RBFM) that you started in project 1. In addition, you will implement a relation manager (RM) on top of the basic record-based file system. Please read the Description of Project 2 in the file CMPS181\_Project2.docx.

**CodeBase**

Same as project 1, we have provided you with a framework to implement project 2. You can download the package from codebaseProject2.zip​. Follow these instructions inside the codebase folder to start your coding:

* Modify the "CODEROOT" variable in makefile.inc to point to the root of your codebase
* Copy your (or Paolo Di Febbo’s) own implementation of record-based file manager (RBFM) component to folder "rbf"
* Implement the remaining methods of the RBFM, and then implement the relation manager (RM):
* You are supposed to implement the API of the RBFM and the RM defined in rbfm.h and rm.h respectively.
* You cannot change those functions of the RBFM and RM classes. If you think some changes are really necessary, please contact us first.

**Memory Requirements**

You should be careful about how to use memory to implement those operations. It is **NOT ACCEPTABLE** to cache the entire database or a large portion of the database in memory, since it is not possible for large amounts of data. For each operation, you should make sure that the "effect" of the operation (if any) has been stored in the file on disk. In case there is a power failure, the data is still on the disk. For example, for the "updateRecord" operation, after the function successfully returns, the updated record should physically reside in the file on the disk.

**Submission Instructions**

The following are requirements on your submission. **Points may be deducted if they are not followed.**

* Write a report to briefly describe the design and implementation of your relation module. Please provide a text file rather than a PDF, Word Document, or other non-text format.
* You need to submit the source code under the "rbf" and "rm" folder. Make sure you do a "make clean" first, and do NOT include any useless files (such as binary files and data files). You should make sure your makefile run properly. We will use the provided test cases to to test your module. In addition we will use private test cases to evaluate your code.
* Please organize your project in the following directory hierarchy, using groupID we will assign soon:  
  project2-*groupID* / codebase / {rm, rbf, makefile.inc, readme.txt, project2-report} where rm and rbf folders include your source code and the makefile.  
  (e.g., project2-01 / codebase / {rm, rbf, makefile.inc, readme.txt, project2-report}
* Compress project2-*groupID* into a SINGLE zip file. Each group only submits one file, with the name "project2-*groupID*.zip". (e.g. project2-01.zip)
* Put testProject2.sh and the zip file under the same directory. Run it to check whether your project can be properly unzipped and tested (use your own makefile.inc and the provided test cases when you are testing the script). If the script doesn't work correctly, it's most likely that your folder organization doesn't meet the requirement. Our grading will be automatically done by running script on the unix.ucsc.edu system, so please test there. The usage of the script is:

./testProject2.sh ''project2-groupID''

* Upload the zip file "project2-*groupID*.zip" to eCommons (not set up yet—will be soon). Only one member of your team should upload the file, not both of you.

**Testing**

Please use the test cases, **rmtest\_create\_tables.cc** and **rmtest\_XX.cc** (where XX is the test case number) included in the codebase to test your code. Note that those files will be used to grade your project partially since we also have our own private test cases. This is by no means an exhaustive test suite. Please feel free to add more cases to this, and test your code thoroughly.

**Important Note:** This time we have provided each test case as a separate program. You must run the test case **rmtest\_create\_tables.cc** first to create the database tables needed by other test cases. Also, you must execute the tests cases in the order provided in testProject2.sh[​](https://grape.ics.uci.edu/wiki/asterix/raw-attachment/wiki/cs222-2014-fall-project2/test.sh). This is because, although the test cases are separate programs, some of the test cases depend on other test cases (e.g., **rmtest\_09.cc** inserts tuples and **rmtest\_10.cc** reads them).

Clearly the more test cases you try, the less likely you will miss bugs in your implementation. Make sure to follow the requirements described in the rbf/rbfm.h and rm/rm.h files (e.g., the format of the data for a record to be read/written from/to the relation manager), since we will write our own test cases to evaluate your implementation (correctness and performance).

Since there is an extra credit of 10 points in this part of the project, we also provided the files **rmtest\_extra\_XX.cc​** (that can be found in the codebase), which can be used to test your code if you decide to implement the extra credit work.

**Grading Rubrics**

Grading rubrics to be added later.

**Q & A**

* **Q1**: Are we allowed to change the rids of existing records in the function reorganizePage() and reorganizeTable()?   
  **A1**: In the function reorganizePage(), 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 records and their slots. However, in the function reorganizeTable(), you have to change the rids, since you need to move records to other pages. Here we assume the caller function will also update those tids in other index structures.
* **Q2**: How should the RM layer lifecycle for file use work - in particular, when would we be expected to open and close RBF layer files and when would we be expected to fetch and update (persist) the on-disk catalogs? Should each RM layer call (such as insertRecord(), readRecord, or scan()) make RBF open/close calls and catalog calls? Please give us a hint as to where the RBF and catalog calls should go.   
  **A2**: (1) Use the RM lifecycle as the open lifecycle for catalog files, i.e. catalog files can be opened in RM constructor, and closed in RM destructor.  
  (2) Catalog can be cached however you'd like to, but you have to make sure it persists across runs. If you do cache metadata, it's probably more convenient to cache it in useful-object form in memory, not pages.  
  (3) For project simplicity, you can open/close data files in each RM layer operation. In future, we may add open/close table functions.
* **Q3**: Assume a file is made of 3 pages, pages 0, 1 and 2. After a few record deletions, page 1 becomes entirely free. Should page 1 be freed up at this point? I notice that there is no deletePage() member function in FileHandle. Does this mean that pages once appended to a file will never be released back until the file is deleted?  
  **A3**: For this case where a page can be completely empty, we have reorganizeFile() method which should take care of such issue.
* **Q4**: Shall we consider the case that a single-page (4096 bytes) can't hold one record, i.e. single record need more than 4096 bytes?  
  **A4**: No. You can always assume that an empty page can hold a record even if a record size is big. In other words, a record size can not exceed the predefined page size.
* **Q5**: Considering project 1 will form the basis for projects 2, 3 and 4, does any part of project 1 need to be thread safe?  
  **A5**: In this project, the assumption is that there is only one user who uses the DBMS system. Therefore, single-threading is a fair assumption here. However, note that there might be multiple accesses to a given file - but only in read-only cases which is consistent with the project description in the wiki: "Warning: Opening a file more than once for data modification is not prevented by the PF component, but doing so is likely to corrupt the file structure and may crash the PF component. Opening a file more than once for reading is no problem."
* **Q6**: In response to a different question, I saw that insertRecord() should result in the changes getting written to the disk. From this, it appears that a functional buffer manager is not required as part of project1. Is this correct? If yes, is it going to be a part of the later projects? Or is it the case that we wont be needing the buffer manager at all for the entire project?  
  **A6**: We have pondered adding a buffer management requirement (w/replacement, etc.) into Part 1, but we've opted to let the Unix filesystem do that instead. (And to just rely on that, and let the layers of this project simply issue reads and writes and appends - letting Unix optimize the reads by finding recent pages in the OS level buffer pool.)