# Project 2 for CS222: Implementing a Relation Manager

**Deadline: Feb 8th, 2019, Saturday, 11:45 pm, on Github.**

## 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 check the details in the Project 2 Description.

## CodeBase

As in Project 1, we have provided you with a framework to implement Project 2. Please download this Project2.patch file, put it in your git repo, and apply it to your codebase with `git am Project2.patch`

OR, if the patch file does not work, you can follow these steps to fetch the code framework manually.

Then you can start Project 2:

- Modify the "CODEROOT" variable in makefile.inc to point to the root of your codebase if necessary.
- Put your own implementation of the record-based file manager (RBFM) component to the folder "rbf".
- Implement the remaining methods of the RBFM, and then implement the relation manager (RM).
- Be sure to implement the API of the RBFM and the RM exactly as defined in rbfm.h and rm.h respectively. If you think changes are necessary, please contact us first.

## Memory Requirements

As you did in Project 1, you should be careful about how you use memory as you implement your project's operations. It is NOT ACCEPTABLE to cache the entire database or even a large portion of the database in memory, since that is not practical for large amounts of data. Also, for each operation, you should make sure that the "effect" of the operation (if any) has indeed been persisted in the appropriate Linux file. For example, for the "insertTuple" operation, after the function successfully returns, the inserted record should have been stored in the file in the Linux filesystem. The tests will help you to not make mistakes here -- that is why each test case is run separately from the others.

## Submission Instructions

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

- Write a report (project2_report.txt) to briefly describe the design and implementation of your relation module by referencing the report template file. 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" folders. Make sure to 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 runs properly. We will use the provided test cases to test your module. In addition we will use a set of private test cases to further evaluate your code, so be aware that you are responsible for turning in well-tested code.

- Please organize your project in the following directory hierarchy:

  master / {rm, rbf, makefile.inc, readme.txt, project2_report.txt, project1_report.txt, CMakeLists.txt}

  where the rm and rbf folders include your source code and the makefile.

- Do NOT modify the makefile.

## Debugging tool Requirements

We highly encourage students to use GDB and Valgrind to debug program. In order to facilitate learning and using GDB and Valgrind, you are required to submit

- One screenshot of debugging your program under Eclipse or other IDE, using GDB or other debuggers. The screenshot must capture the Debug view and have breakpoints in them.
- Two Valgrind reports running on two different test cases of your choice. There may be memory leaks introduced by our provided code, but you should try to get rid of memory leaks as much as you can in your implementation. Grading is based on whether you submit the reports or not and having memory leaks in the reports won't lose the points. The purpose of this requirement is to let you try Valgrind to debug your program.

You can push the screenshots into your master branch along with the project report.txt.

## 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 be sure to test your code thoroughly.

**Important Note:** 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 README.md. This is because, although the test cases are separate programs, some of the test cases depend on other test cases (e.g., **rmtest_08.cc** inserts tuples and **rmtest_09.cc** reads them).

Clearly the more test cases you try, the less likely it is that 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 private test cases to evaluate your implementation (on both correctness and performance).

Since there is an extra credit of 10 points available in this part of the project, we also provided the files **rmtest_extra_XX.cc** (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 are on this page!

## Q & A

- **Q1**: Can I use a WIN32 API like http://msdn.microsoft.com/en-us/library/aa364232%28v=VS.85%29.aspx?
  **A1**: The Windows API will not be allowed. We have provided the MinGW environment, which implements most of the standard lib c/c++ functions on Windows, and those functions are powerful enough for your implementation. Your code should be able to be compiled by a standard g++ compiler. (Implementations based on other OS-specific APIs won't be supported when we do the grading.) Before submitting your project, please make sure that your code can be built by using make with gcc/g++ as the compiler.

- **Q2**: Are we allowed to change the rids of existing records (tuples) when compacting a page after updating or deleting a record (tuple) on it?
  **A2**: You are NOT allowed to change the rids, as they will be used later by other external index structures and it's too expensive to modify those structures for each such change. The rids must not change under any circumstances as long as the corresponding records exist.

- **Q3**: How should the RM layer lifecycle for file use work - in particular, 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 catalog calls? Please give us a hint as to where the catalog calls should go.
  **A3**: Use the RM lifecycle as the open lifecycle for catalog files, i.e. catalog files can be opened in the RM constructor and closed in the RM destructor.

- **Q4**: Suppose that 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?
  **A4**: Yes. Let the future insertions or updates fill that page again.

- **Q5**: Shall we consider the case that a single-page (4096 bytes) can't hold one record, i.e. can a single record need more than 4096 bytes?
  **A5**: No. You can assume that an empty page can hold at least one record even if the record size is "big". If a given record (tuple) is too big for an empty page, return an error when its insertions attempted.

- **Q6**: Considering that Project 1 forms the basis for projects 2, 3 and 4, does any part of project 1 need to be thread safe?
  **A6**: 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."

- **Q7**: 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?
  **A7**: We have periodically pondered adding a buffer management requirement (w/replacement, etc.) into Part 1 - and almost did that for the grad-level project - but we've opted to let the Unix filesystem do that instead. (And to just rely on that, and to therefore 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.)

*Last modified on 2020-1-21 上午10:01:42*

**附件** (1)