Project 2 for CS181: Implementing a Relation Manager

Deadline: Thursday, May 2, 2019, 11:59 pm, on Canvas.

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 CMPS181_Project2_Description file.

CodeBase

As in Project 1, we have provided you with a framework to implement Project 2 using the new codebase.zip file for Project 2 that is on Piazza. 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 if necessary.
- Copy our implementation of the record-based file manager (RBFM) component from Project 1 to the folder "rbf". (Yes, you may use your own implementation if you prefer, but then you will inherit any Project 1 bugs you had in Project 2.)
- 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 very 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, not just in memory. 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 probably be deducted if these instructions are not followed.**

- Write a report to briefly describe the design and implementation of your relation module by filling in the *project2_report.txt* report 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. Be sure to do a "make clean" first; 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 public test cases that we provided 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:
  project2-*teamID* / codebase / {rm, rbf, makefile.inc, readme.txt, project2-report.txt}
  where the rm and rbf folders include your source code and the makefile.
  (E.g., project2-01 / codebase / {rm, rbf, makefile.inc, readme.txt, project2-report.txt}

- Compress project2-*teamID* into a SINGLE zip file. Do not put the string literal - "teamID" in the filename. Each group should only submit one file, with the name "project2-*teamID*.zip". (E.g., project2-01.zip)

- Put the new *test.sh* script provided for Project 2 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 public 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 our requirements. Our grading will be automatically done by running our script. The usage of the script for team 01 (for example) would be:

  ```
  ./test.sh project2-01
  ```

- Upload the zip file *project2-teamID.zip* to Canvas. Only one member of your team should upload the file, not every member of the team.

- NOTE: The entire team is responsible for the upload. So be sure, if your partner does the upload, that you know that the right file has been uploaded on time. You will be graded as a team, so every one of you "owns" the final upload!!

Testing

Please use the test cases **rmtest_create_tables.cc** and **rmtest.cc** included in the codebase to test your code. Note that those files will be used as part of the grading of your project, since we also have private test cases. This is by no means an exhaustive test suite. Please feel free to test your code thoroughly using your own test cases.

Important Note: We have provided each test case as a separate program. But you must run the test case **rmtest_create_tables.cc** first to create the database tables needed by other test cases. Moreover, some of the test cases depend on other test cases, so it's best to always run all of the tests completely, in order.

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 Project 2, we also provided the files **rmtest_extra_1.cc** and **rmtest_extra_2.cc** (which are in the codebase) that can be used to test your code if you decide to implement the extra credit work.

Project 2 Grading Rubric

Full Credit: 100 points
Extra Credit: 10 points

You must follow the following submission requirements in order to receive credit for this assignment. There are no separate points given for this.

- Code is structured as required.
- Makefile works.
- Code compiles and links correctly.

Please make sure that your submission follows these requirements and can be unzipped and built by running the script we provided. You may develop your code on any system using any tools you like, but we will test your submission on unix.ucsc.edu, so you should always test your code on that system. Failure to follow the instructions could make your project unable to build on unix.ucsc.edu, which will result in loss of points.

1. Project Report (10%)

   - Include team numbers, submitter info, and names of other people on your team.
   - Clearly document the design of your project (the record format, the page format, the page format in the file, and the metadata) using the *project2_report.txt* report file template that we provided.
   - Explain the implementation of important functions, such as updateTuple and deleteTuple.

2. Implementation details (15%)

   - Implement all the required functionalities.
   - Implementation should follow the basic requirements of the project. For example, both catalog information and table files should persist on disk. After RM is shut down and restarted, catalog and data should be able to be loaded correctly.

3. Pass the public test suite. (50%)

   - Each of the public tests is graded as pass/fail.

4. Pass the private test suite. (20%)

   - Each of our private tests is graded as pass/fail.

5. Valgrind (5%)

   - You should not have any memory leaks. A valgrind check should not fail.

6. Extra credit work: Implement the following functionalities and pass the provided extra credit test cases: (10%)

- addAttributes/dropAttributes

Note that those methods should internally use the corresponding methods of the record-based file manager.

Q & A

- **Q1**: Are we allowed to change the rids of existing records (tuples) when compacting a page after updating or deleting a record (tuple) on it?
  A1: 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.

- **Q2**: 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.
  A2: 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.

- **Q3**: 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?
  A3: Yes. Let the future insertions or updates fill that page again.

- **Q4**: Should 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?
  A4: 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.

- **Q5**: Considering that Project 1 forms 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 2 Description file: "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."