CMPS 181, Spring 2018, Project 2

Due Thursday, May 3, 11:59pm on Canvas

Course Project 2 Environment and Submission

Introduction

In Project 2, you will continue implementing the record-based file manager (RBFM) that you started in Project 1. As the basis for Project 2, you may use your solution to Project 1, or our Project 1 solution, or a combination of the two. We suggest that you use our solution, so that you don't inherit any of your Project 1 bugs when you do Project 2. 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 file.

CodeBase

As in Project 1, we have provided you with a framework to implement Project 2. You can download the package from Resources→Project2 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 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 avoid 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 be deducted if they are not followed.**

- Write a report to briefly describe the design and implementation of your relation module by modifying the project2_report.txt file that's in codebase. Please be sure to provide that modified 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 ensure that your makefile runs properly on unix.ucsc.edu. 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:
  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., project2-05 / codebase / {rm, rbf, makefile.inc, readme.txt, project2-report.txt}

- Compress project2-*teamID* into a SINGLE zip file. Do not put the string literals - "team" or "teamID" in the filename. Each group should only submit one file, with the name "project2-*teamID*.zip". (E.g., project2-05.zip)
- Put the test.sh file that's in the codebase and your zip file under the same directory. Run test.sh on unix.ucsc.edu 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 our requirements. Our grading will be automatically done by running our script. The usage of the script is:

  ```
  ./test.sh ''project2-teamID''
  ```

- Upload the zip file "project2-*teamID*.zip" to Canvas. Only one member of your team should upload the file, not all of you.
- NOTE: You are all responsible for your team's upload. So be sure, if one of your teammates does it, that you know what has been uploaded and when. You will be graded as a team, not as individuals, so you all "own" the final upload!!

Testing

Please use the test cases **rmtest_create_tables.cc** and **rmtest_XX.cc** (where XX is the test case number) that are included in the codebase to test your code. Note that those files will be used in part to grade your project since we may also have our own private test cases. This is by no means an exhaustive test suite. Please feel free to run your test cases, 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 the test.sh script. 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 overlook bugs in your implementation. Be 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 may write our own private test cases to evaluate your implementation (on both correctness and performance).

Project 2 Grading Rubric

Full Credit: 100 points

You must follow the usual submission requirements in order to receive credit for this assignment, but 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. Failure to follow the instructions could make your project unable to build on our machines, which may result in loss of points.

Note that testing for memory leaks using Valgrind is not required, but it is recommended.

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 meta-data) using the template file that we have provided.
   - Explain the implementation of important functions such as updateTuple and deleteTuple.

2. Implementation details (20%)

   - 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 provided test suite. (50%)

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

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

   - There are private tests, each of which is graded as pass/fail.

Q & A

- **Q1**: Can I develop on a system other than unix.ucsc.edu?
  A1: You may develop on any system that you like, although there are major advantages to developing on unix.ucsc.edu, since that's where we'll test your code. Before submitting your project, please make sure that your code can be built by using make with gcc/g++ as the compiler on unix.ucsc.edu.

- **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 their corresponding records exist.

- **Q3**: Catalogs are relations just like other relations, and they have to be read and updated. How should the Relation Manager (RM) use the catalogs? When is catalog data fetched (read) and when is it updated (persisted to the on-disk catalogs)? Should each RM layer call (such as insertRecord(), readRecord, or scan()) make catalog calls?
  A3: Catalog files provide necessary info about relations, and they need to be read when you need data about a relation and updated/persisted when data about a relation is created (or changed). 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, pages are never released. Let 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: You can assume that an empty page can hold at least one record. If a record (tuple) is too big for an empty page, return an error when its insertion (or update) is attempted.

- **Q6**: Does code for Project 2 need to be thread-safe, since Projects 3 and 4 will be built on it?
  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 could be multiple accesses to a given file -- but only in read-only cases, which is consistent with the project description already given in the Project 1 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. (You do not need to try and prevent this, as you can assume the layer above is "friendly" in that regard.) Opening a file more than once for reading is no problem."

- **Q7**: Is a functional buffer manager required in any part of this Project.
  A7: No. Although understanding frames, dirty bits and pin counts is worthwhile, we've decided to let the Unix filesystem handle all that. Your code simply issue reads and writes and appends, letting Unix optimize the reads by finding recent pages in the OS level buffer pool. In practice, database systems don't do that, but we didn't want to make Projects 1 and 2 overly complicated.