Introduction

In this project, you will implement a paged file (PF) system and the first few operations of a record-based file (RBF) manager. The PF component provides facilities for higher-level client components to perform file I/O in terms of pages. In the PF component, methods are provided to create, destroy, open, and close paged files, to read and write a specific page of a given file, and to add pages to a given file. The record manager is going to be built on top of the basic paged file system. In this part of the project, you are also required to implement some (not all) of the methods provided in the record manager code skeleton.

All methods in the PF and RBF components, except constructors and destructors, return integer codes. A return code of 0 indicates normal completion. A nonzero return code indicates that either an exception condition or an error has occurred.

Interface

The interface of part 1 of the project consists of three classes: the PagedFileManager, the FileHandle, and the RecordBasedFileManager classes.

PagedFileManager Class

The PagedFileManager class handles the creation, deletion, opening, and closing of paged files. Your program should create exactly one instance of this class, and all requests for PF component file management should be directed to that instance. Below, the public methods of the class declaration are shown first, followed by descriptions of the methods. The first two methods in the class declaration are the constructor and destructor methods for the class; they are not explained further. Each method except the constructor and destructor methods returns a value of type RC (for "return code" -- actually an integer). A return code of 0 indicates normal completion. A nonzero return code indicates that an exception condition or error has occurred.

```
class PagedFileManager {
public:

    static PagedFileManager &instance();                                // Access to the _pf_manager instance

    RC createFile(const std::string &fileName);                         // Create a new file
    RC destroyFile(const std::string &fileName);                        // Destroy a file
    RC openFile(const std::string &fileName, FileHandle &fileHandle);   // Open a file
    RC closeFile(FileHandle &fileHandle);                               // Close a file

protected:
    PagedFileManager();                                                 // Prevent construction
    ~PagedFileManager();                                                // Prevent unwanted destruction
    PagedFileManager(const PagedFileManager &);                         // Prevent construction by copying
    PagedFileManager &operator=(const PagedFileManager &);              // Prevent assignment

};
```

RC createFile (const string &fileName)

This method creates an empty-paged file called fileName. The file should not already exist. This method should not create any pages in the file.

RC destroyFile (const string &fileName)

This method destroys the paged file whose name is fileName. The file should already exist.

RC openFile (const string &fileName, FileHandle &fileHandle)

This method opens the paged file whose name is fileName. The file must already exist (and been created using the createFile method). If the open method is successful, the fileHandle object whose address is passed in as a parameter now becomes a "handle" for the open file. This file handle is used to manipulate the pages of the file (see the FileHandle class description below). It is an error if fileHandle is already a handle for some open file when it is passed to the openFile method. It is not an error to open the same file more than once if desired, but this would be done by using a different fileHandle object each time. Each call to the openFile method creates a new "instance" of FileHandle. 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.

RC closeFile (FileHandle &fileHandle)

This method closes the open file instance referred to by fileHandle. (The file should have been opened using the openFile method.) All of the file's pages are flushed to disk when the file is closed.

FileHandle Class

The FileHandle class provides access to the pages of an open file. To access the pages of a file, a client first creates an instance of this class and passes it to the PagedFileManager::openFile method described above. As before, the public methods of the class declaration are shown first, followed by descriptions of the methods. The first two methods in the class declaration are the constructor and destructor methods and are not explained further. In order for you to gather performance data about file accesses, the FileHandle class should maintain counters for each of the I/O-related operations of theFileHandle class. When initialized, all counters should be set to 0. Inside each method of FileHandle, you will then need to increase the corresponding counter by 1 every time you successfully execute the associated method. For instance, when readPage() is executed, readPageCounter should be increased by 1. Some requirements:

1. The counter values binds to a file, and they need to be persistent between multiple file open and close operations of the PagedFileManager.
2. You need to store the counter values physically in a file, and you can decide how to implement it. For example, you can reserve the file's first page as a hidden page to store the counter values as well as the number of pages.
3. You are **NOT** required to handle multiple fileHandles operating on the same file concurrently. Concurrency is not the scope of this course.

```
class FileHandle {
public:
    // variables to keep the counter for each operation
    unsigned readPageCounter;
    unsigned writePageCounter;
    unsigned appendPageCounter;

    FileHandle();                                                    // Default constructor
    ~FileHandle();                                                   // Destructor
```

```
    RC readPage(PageNum pageNum, void *data);                           // Get a specific page
    RC writePage(PageNum pageNum, const void *data);                    // Write a specific page
    RC appendPage(const void *data);                                    // Append a specific page
    unsigned getNumberOfPages();                                        // Get the number of pages in the file
    RC collectCounterValues(unsigned &readPageCount, unsigned &writePageCount,
                            unsigned &appendPageCount);                 // Put current counter values into variables
};
```

RC readPage(PageNum pageNum, void *data)

This method reads the page into the memory block pointed to by data. The page should exist. Note that page numbers start from 0.

Here is a part of some example code for readPage that increases readPageCount whenever it is executed. For writePage() and appendPage(), the logic is similar.

```
    RC FileHandle::readPage(PageNum pageNum, void *data) {
        ......
        readPageCount = readPageCount + 1;
        return 0;
    }
```

RC writePage(PageNum pageNum, const void *data)

This method writes the given data into a page specified by pageNum. The page should exist. Page numbers start from 0.

RC appendPage(const void *data)

This method appends a new page to the end of the file and writes the given data into the newly allocated page.

unsigned getNumberOfPages()

This method returns the total number of pages currently in the file.

RC collectCounterValues(unsigned &readPageCount, unsigned &writePageCount, unsigned &appendPageCount)

This method should return the current counter values of this FileHandle in the three given variables. Here is some example code that gives you an idea how it will be applied.

```
        ......
        unsigned readPageCount = 0;
        unsigned writePageCount = 0;
        unsigned appendPageCount = 0;

        ......
```

```
        FileHandle fileHandle1;
        rc = pfm.openFile(fileName, fileHandle1);
        fileHandle1.collectCounterValues(readPageCount, writePageCount, appendPageCount);
        std::cout << "Before AppendPage - R:" << readPageCount << " W:" << writePageCount << " A:" << appendPageCount

        ......
        rc = fileHandle1.appendPage(data);
        fileHandle1.collectCounterValues(readPageCount, writePageCount, appendPageCount);
        std::cout << "After AppendPage R:" << readPageCount << " W:" << writePageCount << " A:" << appendPageCount <<
```

And a result might be:

```
        Before AppendPage - R:0 W:0 A:0
        After AppendPage - R:0 W:0 A:1
```

In this example, FileHandle appends one page (A + 1).

RecordBasedFileManager Class

The RecordBasedFileManager class handles record-based operations such as inserting, updating, deleting, and reading records. Your program should create exactly one instance of this class, and all requests for this component should be directed to that instance. Below, the public methods of the class declaration are shown first, followed by descriptions of the methods. The first two methods in the class declaration are the constructor and destructor methods for the class; they are not explained further. Each method except the constructor and destructor methods returns a value of type RC. A return code of 0 indicates normal completion. A nonzero return code indicates that an exception condition or error has occurred. Please note that in this part of the project, you are only responsible for implementing the first group of methods in this class (besides the constructor and destructor). Note that for part 1 of the project, you are thus NOT required to implement the following methods: deleteRecord, updateRecord, readAttribute, and scan.

```
class RecordBasedFileManager {

public:
    // Access to the _rbf_manager instance
    static RecordBasedFileManager& instance();

    // Create a new record-based file
    RC createFile(const string &fileName);

    // Destroy a record-based file
    RC destroyFile(const string &fileName);

    // Open a record-based file
    RC openFile(const string &fileName, FileHandle &fileHandle);

    // Close a record-based file
    RC closeFile(FileHandle &fileHandle);
```

```
    // Insert a record into a file
    RC insertRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const void *data, RID &rid);

    // Read a record identified by the given rid.
    RC readRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid, void *data);

    // Print the record that is passed to this utlity method.
    RC printRecord(const vector<Attribute> &recordDescriptor, const void *data);


/****************************************************************************************************************

    * IMPORTANT, PLEASE READ: All methods below this comment (other than the constructor and destructor) are NOT
required to be implemented for Project 1 *

****************************************************************************************************************


    // Delete a record identified by the given rid.
    RC deleteRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid);

    // Update a record identified by the given rid.
    RC updateRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const void *data, const RID
&rid);

    // Read an attribute given its name and the rid.
    RC readAttribute(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid, const string
attributeName, void *data);

    // Scan returns an iterator to allow the caller to go through the results one by one.
    RC scan(FileHandle &fileHandle,
      const vector<Attribute> &recordDescriptor,
      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
      RBFM_ScanIterator &rbfm_ScanIterator);

protected:
    RecordBasedFileManager();                                                     // Prevent construction
    ~RecordBasedFileManager();                                                    // Prevent unwanted destruction
    RecordBasedFileManager(const RecordBasedFileManager &);                      // Prevent construction by copying
    RecordBasedFileManager &operator=(const RecordBasedFileManager &);           // Prevent assignment

}
```

RC createFile(const string &fileName)

This method creates a record-based file called fileName. The file should not already exist. Please note that this method should internally use the method PagedFileManager::createFile (const char *fileName).

RC destroyFile(const string &fileName)

This method destroys the record-based file whose name is fileName. The file should exist. Please note that this method should internally use the method PagedFileManager::destroyFile (const char *fileName).

RC openFile(const string &fileName, FileHandle &fileHandle)

This method opens the record-based file whose name is fileName. The file must already exist and it must have been created using the RecordBasedFileManager::createFile method. If the method is successful, the fileHandle object whose address is passed as a parameter becomes a "handle" for the open file. The file handle rules in the method PagedFileManager::openFile apply here too. Also note that this method should internally use the method PagedFileManager::openFile(const char *fileName, FileHandle &fileHandle).

RC closeFile(FileHandle &fileHandle)

This method closes the open file instance referred to by fileHandle. The file must have been opened using the RecordBasedFileManager::openFile method. Note that this method should internally use the method PagedFileManager::closeFile(FileHandle &fileHandle).

RC insertRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const void *data, RID &rid)

Given a record descriptor, insert a new record into the file identified by the provided handle. You can assume that the input is always correct and free of error. That is, you do not need to check to see if the input record has the right number of attributes or if the attribute types match. However, in order to deal with NULL values in the attributes, the first part in *data contains n bytes for passing the null information about each fields. The value n can be calculated by using this formula: ceil(number of fields in a record / 8). For example, if there are 5 fields, ceil(5/8) = 1 byte. If there are 20 fields, the size will be ceil(20/8) = 3 bytes. The left-most bit in the first byte corresponds to the first field. The right-most bit in the first byte corresponds to the eighth field. If there are more than eight fields, the left-most bit in the second byte corresponds to the ninth field and so on. If the corresponding bit to each field is set to 1, then the actual data does not contain any value for this field. For example, if there are three fields in a record and the second field contains NULL, the bit representation in a byte is 0100000. In addition, in the actual data, the incoming record contains the first and the third values only. That is, the third field value is placed right after the first field value in this case.

This format (null-fields-indicator + actual data) is to be used for all record manipulation operations (unless stated differently). For example, when you read a record, the first part of what you return should contain a null-fields-indicator that provides the information about null fields, and the actual data should not contain null field values.

Please read the source code to get more details about the format of "*data", including how each attribute is represented as bytes.

Your file structure is a heap file, and you may use a system-sequenced file organization. That is, if the last (current) page has enough space, insert a new record into this page. If not, find the first page with free space large enough to store the record, e.g., looking from the beginning of the file, and store the record at that location. An RID here is the record ID which is used to uniquely identify records in a file. An RID consists of: 1) the page number that the record resides in within the file, and 2) the slot number that the record resides in within the page. The insertRecord method accepts an RID object and fills it with the RID of the record that is the target for insertion; this lets the caller know what the system-determined RID was for each newly inserted record. For managing free space within pages, you should keep the free space coalesced in the center

of the page at all times -- so if a record deletion or update creates a "hole", you should move records around to keep all of the free space together. Note that this will not change the RIDs of your records; when you move a record within a page, you will also keep track of the record's new offset in the slot table on the page, and the RID only contains the slot number, not the offset itself.

Note that the API data format above is just intended for passing the data into the insertRecord(). This does not necessarily mean that the internal representation of your record should be the same as this format. (It probably shouldn't be. :-)) **Your record representation must allow direct addressability of data fields - i.e., finding the nth field must be an O(1) operation, not an O(n) operation.**

RC readRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid, void *data)

Given a record descriptor, read the record identified by the given rid.

RC printRecord(const vector<Attribute> &recordDescriptor, const void *data)

To ease grading, please make sure you print as the required format:

`<AttributeName1>:\s<Value1>\s<AttributeName2>:\s<Value2>\s<AttributeName3>:\s<Value3>\n`

This is a utility method that will be mainly used for debugging/testing. It should be able to interpret the bytes of each record using the passed-in record descriptor and then print its content to the screen. For instance, suppose a record consists of two fields: age (int) and height (float), which means the record will be of size 9 (1 byte for the null-fields-indicator, 4 bytes for int, and 4 bytes for float). The printRecord method should recognize the record format using the record descriptor. It should then check the null-fields-indicator to skip certain fields if there are any NULL fields. Then, it should be able to convert the four bytes after the first byte into an int object and the last four bytes to a float object and print their values. It should also print NULL for those fields that are skipped because they are null. Thus, an example for three records would be:

```
age: 24      height: 6.1
age: NULL    height: 7.5
age: 32      height: NULL
```

RC deleteRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid)

Given a record descriptor, delete the record identified by the given rid. Also, each time when a record is deleted, you will need to compact that page. That is, keep the free space in the middle of the page -- the slot table will be at one end, the record data area will be at the other end, and the free space should be in the middle. Also, if a slot is deleted, it should be reused by the next insertRecord (rather than leaving a hole there and never reuse it).

RC updateRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const void *data, const RID &rid)

Given a record descriptor, update the record identified by the given rid with the passed data. If the record grows and there is no space in the page to store the record, the record must be migrated to a new page with enough free space. Since you will soon be implementing an index structure, assume that records are identified by their RID values and when they migrate, you should leave a tombstone behind (pointing to the new location of the record). Also, each time when a record is updated to become smaller, you need to compact that page. That is, keep the free space in the middle of the page -- the slot table will be at one end, the record data area will be at the other end, and the free space should be in the middle. Again, the structure for *data is the same as the one we use for the insertRecord().

RC readAttribute(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid, const string attributeName, void *data)

Given a record descriptor, read a specific attribute of a record identified by a given rid.

RC scan(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const string &conditionAttribute, const CompOp compOp, const void *value, const vector<string> &attributeNames, RBFM_ScanIterator &rbfm_ScanIterator)

Given a record descriptor, scan a file, i.e., sequentially read all the entries in the file. 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"). Specifically, the parameter conditionAttribute here is the attribute's name that you are going to apply the filter on. The compOp parameter is the comparison type that is going to be used in the filtering process. **The value parameter is the value of the conditionAttribute that is going to be used to filter out records. For INT and REAL, the value is simply stored as 4 bytes. For VARCHAR, the value contains 4 bytes for the length of the characters followed by the actual characters.** Note that the retrieved records should only have the fields that are listed in the vector attributeNames. Please take a look at the test cases for more information on how to use this method.

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 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 stored in the Linux file. For example, for the "insertRecord" operation, after the function successfully returns, the inserted record should physically reside 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.

Record Representation

- You need to support basic attribute types, including integers, reals, and variable-length character strings. Other types are optional.
- There can be NULL values in one or multiple fields in a record, as described throughout this document.
- You can assume that the float value comparisons can be done by using logical operators (==, !=, >=, <=, <, >).
- Endianness is not an issue here since you do not communicate with other systems. Use the default setting that your system provides.
- Records within file pages should be represented using a record format that nicely handles mixes of binary data and variable-length character data. "Nicely" here refers to both space and efficiency, e.g., you should not waste 70 bytes of space to store "abcdefghij" in a VARCHAR(80) field.
- **Your record representation must allow direct addressability of data fields - i.e., finding the nth field must be an O(1) operation, not an O(n) operation.**
- Your chosen on-page record format should be clearly documented in your project code and your accompanying report.

Last modified on 2020-1-7 上午10:07:22