**CMPS 181, Spring 2015: Project 1 Introduction**

In this project, you will implement a paged file (PF) system and 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 requested to implement some (and 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 an integer code. 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:

// Access to the \_pf\_manager instance

static PagedFileManager\* instance();

// Create a new file

RC createFile (const char \*fileName);

// Destroy a file

RC destroyFile (const char \*fileName);

// Open a file

RC openFile (const char \*fileName, FileHandle &fileHandle);

// Close a file

RC closeFile (FileHandle &fileHandle);

protected:

// Constructor

PagedFileManager();

// Destructor

~PagedFileManager();

private:

static PagedFileManager\* \_pf\_manager;

}

**RC createFile (const char \*fileName)**

This method creates a paged file called fileName. The file should not already exist.

**RC destroyFile (const char \*fileName)**

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

**RC openFile (const char \*fileName, FileHandle &fileHandle)**

This method opens the paged file whose name is fileName. The file must already exist and it must have been created using the 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 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 an 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, using a different fileHandle object each time. Each call to the openFile method creates a new "instance" of the open 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.

**RC closeFile (FileHandle &fileHandle)**

This method closes the open file instance referred to by fileHandle. The file must 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.

class FileHandle {

public:

// Constructor

FileHandle();

// Destructor

~FileHandle();

// Get a specific page

RC readPage(PageNum pageNum, void \*data);

// Write a specific page

RC writePage(PageNum pageNum, const void \*data);

// Append a specific page

RC appendPage(const void \*data);

// Get the number of pages in the file

unsigned getNumberOfPages();

}

**RC readPage(PageNum pageNum, void \*data)**

This method reads the page into the memory block pointed by data. The page should exist. Note the page number starts from 0.

**RC writePage(PageNum pageNum, const void \*data)**

This method writes the data into a page specified by the pageNum. The page should exist. Note the page number starts from 0.

**RC appendPage(const void \*data)**

This method appends a new page to the file, and writes the data into the new allocated page.

**unsigned getNumberOfPages()**

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

**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 of implementing three methods in this class (besides the constructor and destructor). Note that for part 1 of the project, you are NOT required to implement the following methods: deleteRecords, deleteRecord, updateRecord, readAttribute, reorganizePage, scan, and reorganizeFile.

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);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* All methods below this comment (other than the constructor and destructor) are NOT required to be implemented for part 1 of the project

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Delete all records in a file.

RC deleteRecords(FileHandle &fileHandle);

// 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);

// Push the free space towards the end of the page.

RC reorganizePage(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const unsigned pageNumber);

// 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);

// Reorganize the records in the file such that the records are collected towards the beginning of the file.

RC reorganizeFile(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor);

protected:

// Constructor

RecordBasedFileManager();

// Destructor

~RecordBasedFileManager();

private:

static RecordBasedFileManager\* \_rbf\_manager;

}

**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 applies 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 record into a given file identifed 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 if the input record has the right number of attributes and if the attribute types match. Furthermore, you may use system-sequenced file organization. That is, find the first page with free space large enough to store the record and store the record at that location. 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 target for insertion.

**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)**

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 record descriptor and then print its content to the screen. For instance suppose a record consists of two fields: int and float, Which means the record will be of size 8 (ignoring any other metadata information that might be kept with every record). The printRecord method will be able to recognize the record format using the record descriptor. It then will be able to convert the first four bytes to an int object and the last four bytes to a float object and print their values.

**RC deleteRecords(FileHandle &fileHandle)**

Delete all records in the file.

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

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

**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 is migrated to a new page with enough free space. Since you will soon be implementing a B-tree structure or any indexing mechanism, assume that records are identified by their rids and when they migrate, they leave a tombstone behind pointing to the new location of the record.

**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 reorganizePage(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const unsigned pageNumber)**

Given a record descriptor, reorganize a page, i.e., push the free space towards the end of the page.

**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. 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.

**RC reorganizeFile(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor)**

Given a record descriptor, reorganize the file which causes reorganization of the records such that the records are collected towards the beginning of the file. Also, record redirection is eliminated. (In this case, and only this case, it is okay for rids to change.)

**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 "insertRecord" operation, after the function successfully returns, the inserted record should physically reside in the file on the disk.

**Record Representation**

* You need to support basic attribute types, including integers, reals, and variable-length character strings. Other types are optional.
* Records within file pages should be represented using a record format that nicely handles mixes of binary data and variable-length 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. **Please note that this requirement is optional. Students that implement this feature will get an extra credit of 5 points.**
* Your record representation should 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 format should be clearly documented in your project code.