CS564 Fall 2014
Assignment 4 - The Heap File

Due: 11:30PM on Sunday, 23rd November 2014
Individual Assignment for Epic Section
At most 2 people in a team for On-Campus Section

Introduction

In this part of the project, you will implement a File Manager for Heap Files that also provides a scan mechanism that will allow you to search a heap file for records that satisfy a search predicate called a *filter*. How are heap files different from the files that the DB layer of Minirel provides? The main difference is that the files at the DB layer are physical (a bunch of pages) and not logical. The HeapFile layer imposes a logical ordering on the pages (via a linked list).

Classes

The HeapFile layer contains three main classes: a **FileHdrPage** class that implements a heap file using a linked list of pages, a **HeapFile** class for inserting and deleting records from a heap file and a **HeapFileScan** class (derived from the HeapFile class) for scanning a heap file (with an optional predicate). The detailed description of each of these classes follows.

The FileHdrPage Class

Each heap file consists of one instance of the FileHdrPage class plus one or more data pages. This header page is different from the DB header page for the underlying physical file provided by the I/O layer.

```
class FileHdrPage
{
        char fileName[MAXNAMESIZE];    // name of file
        int  firstPage;                // pageNo of first data page in file
        int  lastPage;                 // pageNo of last data page in file
        int  pageCnt;                  // number of pages
        int  recCnt;                   // record count
};
```

The meaning of most of the fields in this structure should be obvious from their names. lastPage is used to keep track of the last page in the file. The firstPage attribute of the header page points to the first data page. The data pages are instances of the Page class and they are linked together using a linked list (via the nextPage data member).

The FileHdrPage class has NO member functions. The reason for this will be explained below.

Associated with this class are two functions: **createHeapFile()** and **destroyHeapFile().** You have to to implement createHeapFile()

```
const Status createHeapFile(const string filename)
```

You have to implement this method.

This function creates an empty (well, almost empty) heap file. To do this create a db level file by calling db->createFile(). Then, allocate an empty page by invoking bufMgr->allocPage() appropriately. As you know allocPage() will return a pointer to an empty page in the buffer pool along with the page number of the page. Take the Page* pointer returned from allocPage() and cast it to a FileHdrPage*. Using this pointer initialize the values in the header page. Then make a second call to bufMgr->allocPage(). This page will be the first data page of the file. Using the Page* pointer returned, invoke its init() method to initialize the page contents. Finally, store the page number of the data page in firstPage and lastPage attributes of the FileHdrPage.

When you have done all this unpin both pages and mark them as dirty.

```
const Status destroyHeapFile (const string filename)
```

This is easy. Simply calls db->destroyFile(). The user is expected to have closed all instances of the file before calling this function.

The HeapFile Class

As discussed above, a heap file consists of one file header page and 1 or more data pages. The HeapFile class provides a set of methods to manipulate heap files including adding and deleting records and scanning all records in a file. Creating an instance of the heapFile class opens the heap file and reads the file header page and the first data page into the buffer pool.

Here is the definition of the HeapFile class:

```
class HeapFile {
        protected:
                File*         filePtr;        // underlying DB File object
                FileHdrPage*  headerPage;     // pinned file header page in buffer pool
                int           headerPageNo;   // page number of header page
                bool          hdrDirtyFlag;   // true if header page has been updated

                Page*    curPage;         // data page currently pinned in buffer pool
                int      curPageNo;       // page number of pinned page
                bool     curDirtyFlag;    // true if page has been updated
                RID      curRec;          // rid of last record returned

        public:
                // initialize
                HeapFile(const string& name, Status& returnStatus);

                // destructor
                ~HeapFile();

                // return number of records in file
                const int getRecCnt() const;

                // given a RID, read record from file, returning pointer and length
                const Status getRecord(const RID& rid, Record& rec);
};
```

The public member functions are described below.

`HeapFile(const string& name, Status& returnStatus)`

You have to implement this method.

This method first opens the appropriate file by calling db->openFile() (do not forget to save the File* returned in the filePtr data member). Next, it reads and pins the header page for the file in the buffer pool, initializing the private data members headerPage, headerPageNo, and hdrDirtyFlag. You might be wondering how you get the page number of the header page. This is what file->getFirstPage() is used for (see description of the I/O layer)! Finally, read and pin the first page of the file into the buffer pool, initializing the values of curPage, curPageNo, and curDirtyFlag appropriately. Set curRec to NULLRID.

`~HeapFile()`

The destructor first unpins the header page and currently pinned data page and then calls db->closeFile()

`const int getRecCnt() const`

Returns the number of records currently in the file (as found in the header page for the file).

`const Status getRecord (const RID& rid, Record& rec)`

You have to implement this method.

This method returns a record (via the rec structure) given the RID of the record. The private data members curPage and curPageNo should be used to keep track of the current data page pinned in the buffer pool. If the desired record is on the currently pinned page, simply invoke curPage->getRecord(rid, rec) to get the record. Otherwise, you need to unpin the currently pinned page (assuming a page is pinned) and use the pageNo field of the RID to read the page into the buffer pool

The HeapFileScan Class

The HeapFile class primarily deals with pages.To insert records into a file, the InsertFileScan class is used (described below). To retrieve records the HeapFileScan class (which also represents a HeapFile since it is derived from the HeapFile class) is used. The HeapFileScan class can be used in three ways:

1. To retrieve all records from a HeapFile.
2. To retrieve only those records that match a specified predicate.
3. To delete records in a file.

Several HeapFileScans may be instantiated on the same file simultaneously.This will work fine as long as each has its own "current" FileHdrPage pinned in the buffer pool. Note that the HeapFileScan class is derived from the HeapFile class, thus inheriting its data members and member functions.

```
class HeapFileScan : public HeapFile {
        public:
                HeapFileScan(const string& name, Status& status);

                // end filtered scan
                ~HeapFileScan();

                const Status startScan(const int offset, const int length, const Datatype type, const char* filter, const Operator op);

                const Status endScan(); // terminate the scan

                const Status markScan(); // save current position of scan

                const Status resetScan(); // reset scan to last marked location

                // return RID of next record that satisfies the scan
                const Status scanNext(RID& outRid);

                // read current record, returning pointer and length
                const Status getRecord(Record& rec);

                // delete current record
                const Status deleteRecord();

                // marks current page of scan dirty
                const Status markDirty();

        private:
                int   offset;            // byte offset of filter attribute
                int   length;            // length of filter attribute
                Datatype type;           // datatype of filter attribute
                const char* filter;      // comparison value of filter
                Operator op;             // comparison operator of filter

                // The following variables are used to preserve the state
                // of the scan when the method markScan() is invoked.
                // A subsequent invocation of resetScan() will cause the
                // scan to be rolled back to the following
                int   markedPageNo; // value of curPageNo when scan was "marked"

                RID   markedRec;    // value of curRec when scan was "marked"

                const bool matchRec(const Record& rec) const;
};
```

Private Data Members of HeapFileScan:

First note that the protected data members curPage, curPageNo, curRec, and curDirtyFlag should be used to implement the scan mechanism.

The other private data members are used to implement conditional or filtered scans. length is the size of the field on which predicate is applied and offset is its position within the record (we consider fixed length attributes only). The type of the attribute can be STRING, INTEGER or FLOAT and is defined in heapFile.h Similarly all ordinary comparison operators (as defined in heapFile.h) must be supported. The value to be compared against is stored in *binary* form and is pointed to by filter.

`HeapFileScan(const string& name, Status& status)`

Initializes the data members of the object and then opens the appropriate heapfile by calling the HeapFile constructor with the name of the file. The status of all these operations is indicated via the status parameter.

For those of you new to C++, since HeapFileScan is derived from HeapFile, the constructor for the HeapFile class is invoked before the HeapFileScan constructor is invoked.

`~HeapFileScan()`

Shuts down the scan by calling endScan(). After the HeapFileScan destructor is invoked, the HeapFile destructor will be automatically invoked

`const Status startScan(const int offset, const int length, const Datatype type, const char* filter, const Operator op)`

This method initiates a scan over a file. If filter == NULL, an unconditional scan is performed meaning that the scan will return all the records in the file. Otherwise, the data members of the HeapFileScan object are initialized with the parameters to the method.

`const Status endScan()`

This method terminates a scan over a file but does not delete the scan object. This will allow the scan object to be reused for another scan.

`const Status markScan()`

Saves the current position of the scan by preserving the values of curPageNo and curRec in the private data members markedPageNo and markedRec, respectively.

`const Status resetScan()`

Resets the position of the scan to the position when the scan was last marked by restoring the values of curPageNo and curRec from markedPageNo and markedRec, respectively. Unless the page number of the currently pinned page is the same as the marked page number, unpin the currently pinned page, then read markedPageNo from disk and set curPageNo, curPage, curRec, and curDirtyFlag appropriately.

`const Status scanNext(RID& outRid)`

You have to implement this method.

Returns (via the outRid parameter) the RID of the next record that satisfies the scan predicate. The basic idea is to scan the file one page at a time. For each page, use the firstRecord() and nextRecord() methods of the Page class to get the rids of all the records on the page. Convert the rid to a pointer to the record data and invoke matchRec() to determine if record satisfies the filter associated with the scan. If so, store the rid in curRec and return curRec. To make things fast, keep the current page pinned until all the records on the page have been processed. Then continue with the next page in the file. Since the HeapFileScan class is derived from the HeapFile class it also has all the methods of the HeapFile class as well. Returns OK if no errors occurred. Otherwise, return the error code of the first error that occurred.

`const Status getRecord(Record& rec)`

Returns (via the rec parameter) the record whose rid is stored in curRec. The page containing the record should already be pinned in the buffer pool (by a preceding scanNext() call). If not, return BADPAGENO. A pointer to the pinned page can be found in curPage. Just invoke Page::getRecord() on this page and return its status value.

`const Status deleteRecord()`

Deletes the record with RID whose rid is stored in curRec from the file by calling Page::deleteRecord. The record must be a record on the "current" page of the scan. Returns BADPAGENO if the page number of the record (i.e. curRec.pageNo) does not match curPageNo, otherwise OK.

`const bool matchRec(const Record& rec) const`

This **private** method determines whether the record rec satisfies the predicate associated with the scan. It takes care of attributes not being aligned properly. Returns *true* if the record satisfies the predicate, *false* otherwise. **We will provide this method for you.**

`const Status markDirty()`

Marks the current page of the scan dirty by setting the dirtyFlag. The dirtyFlag should be reset every time a page is pinned. Make sure that when you unpin a page in the buffer pool, that you pass dirtyFlag as a parameter to the unpin call. Returns OK if no errors. The page containing the record should already be pinned in the buffer pool.

The InsertFileScan Class

The InsertFileScan Class is used to insert records into a file. Like the HeapFileScan class it is derived from the HeapFile class. Its definition is given below:

```
class InsertFileScan : public HeapFile {
        public:
                InsertFileScan(const string& name, Status& status);

                // end filtered scan
                ~InsertFileScan();

                // insert record into file, returning its RID
                const Status insertRecord(const Record& rec, RID& outRid);
};
```

`InsertFileScan(const string& name, Status& status)`

Again this constructor will get invoked after the HeapFile constructor is invoked since InsertFileScan is derived from the HeapFile class. See the test program for examples of how this constructor is used.

`~InsertFileScan()`

Unpins the last page of the scan and then returns. The HeapFile destructor will be automatically invoked.

`const Status insertRecord (const Record& rec, RID& outRid)`

You have to implement this method.

Inserts the record described by rec into the file returning the RID of the inserted record in outRid. First, check to see if the current page is the last page of the file. If it is, try to insert into the current page. If the last page is full then allocate a new page and insert the record into the new page (make sure that you update the header page appropriately; you also have to make the newly allocated page into the current page; further, you have to pin and unpin pages appropriately).

If the current page is not the last page, bring the last page of the file into the buffer pool, make it the current page and follow the same procedure.

Getting Started

Start by making a copy of heapfile.tar.gz from the Assignment 4 directory in Learn@UW. In it you will find the following files:

- `Makefile` - A make file. You can make the project by typing `make'
- `buf.h` - Class definitions for the buffer manager. You should not change these
- `buf.cpp` - Implementations of the methods. You should not change these
- `bufHash.cpp` - Implementation of the buffer pool hash table class. You should not change these
- `db.h` - Class definitions for the DB and File classes. You should not change these
- `db.cpp` - Implementations of the DB and File classes. You should not change these
- `page.cpp` - Implementation of the page class. You should not change these
- `page.h` - Class definition of the page class. You should not change these
- `error.h` - Error codes and error class definition
- `error.cpp` - Error class implementation
- `heapfile.h` - Definitions for the HeapFile and HeapFileScan classes.
- `heapfile.cpp` - Skeleton implementations for the above classes. *You should complete these*
- `testfile.cpp` - Test driver. Feel free to augment this if you want

Handing In

To handin the stage of your project:

- Follow the same steps as the previous assignment and upload you tar file to the *Assignment 4 Submission* dropbox folder on Learn@UW.
- Please remember it is due at 11:30PM on Sunday, 23rd November 2014. The late policy will be as mentioned here. If you do not have the assignment complete by that time, please turn in whatever you have in hopes of getting partial credit.

Error Checking

This is some additional clarification about how error checking should be done. Each function that you implement must do some error checking on its input parameters to make sure that they are valid. If they are not, then the function should return an appropriate error code. Each function should also check the return status of every function it calls. If an error occurred, you should just return this code to the higher level calling function. Remember to do any necessary cleanup while doing a premature exit from a function. Usually, you should not print out error messages in any of the functions you implement. These should only be printed by the top level calling function in testfile.cpp for example. The intermediate functions in the HeapFile and Buffer Manager layers should just relay the error codes to the higher layers.