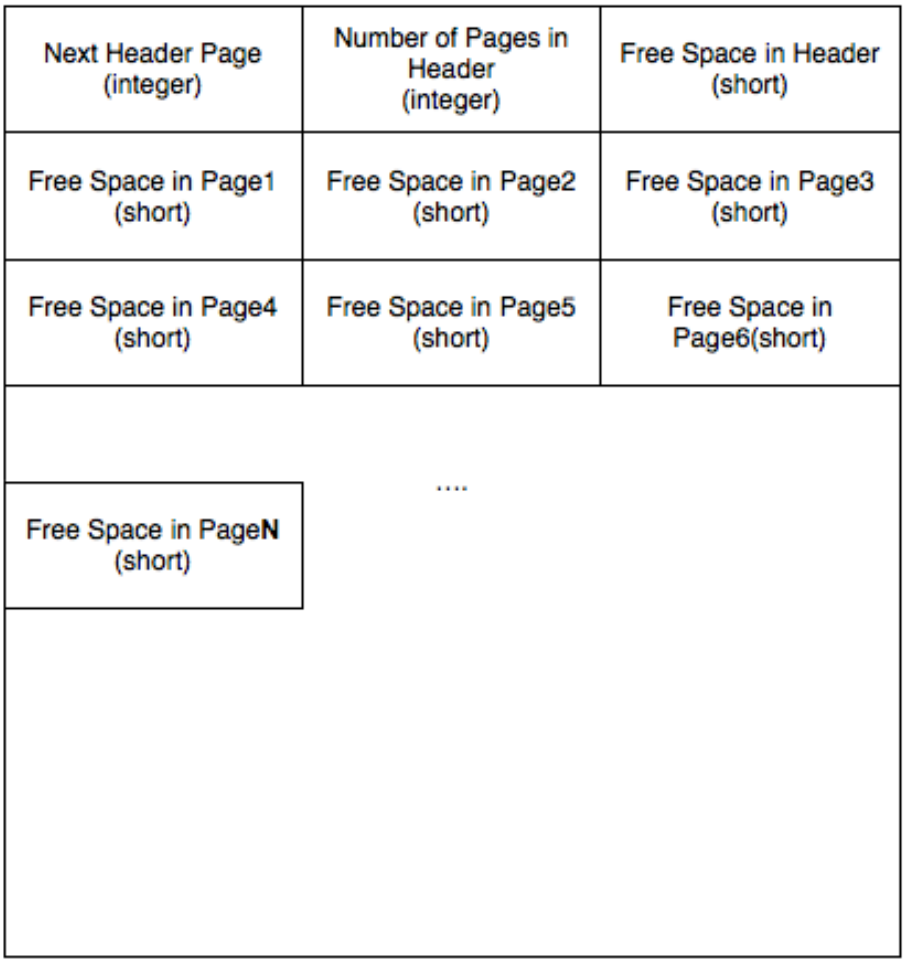
Liangjun Feng
John Palomo
Group ID: 15

The following report describes the design decisions that we considered when implementing the paged file manager and record based file management. The report is not exhaustive, but will touch upon those decisions that may deviate from other implementations.

Before explaining the changes made to each respective class, it is important to understand the structure we defined to represent header pages and data entry pages.

Header Page Structure



Header Page Layout

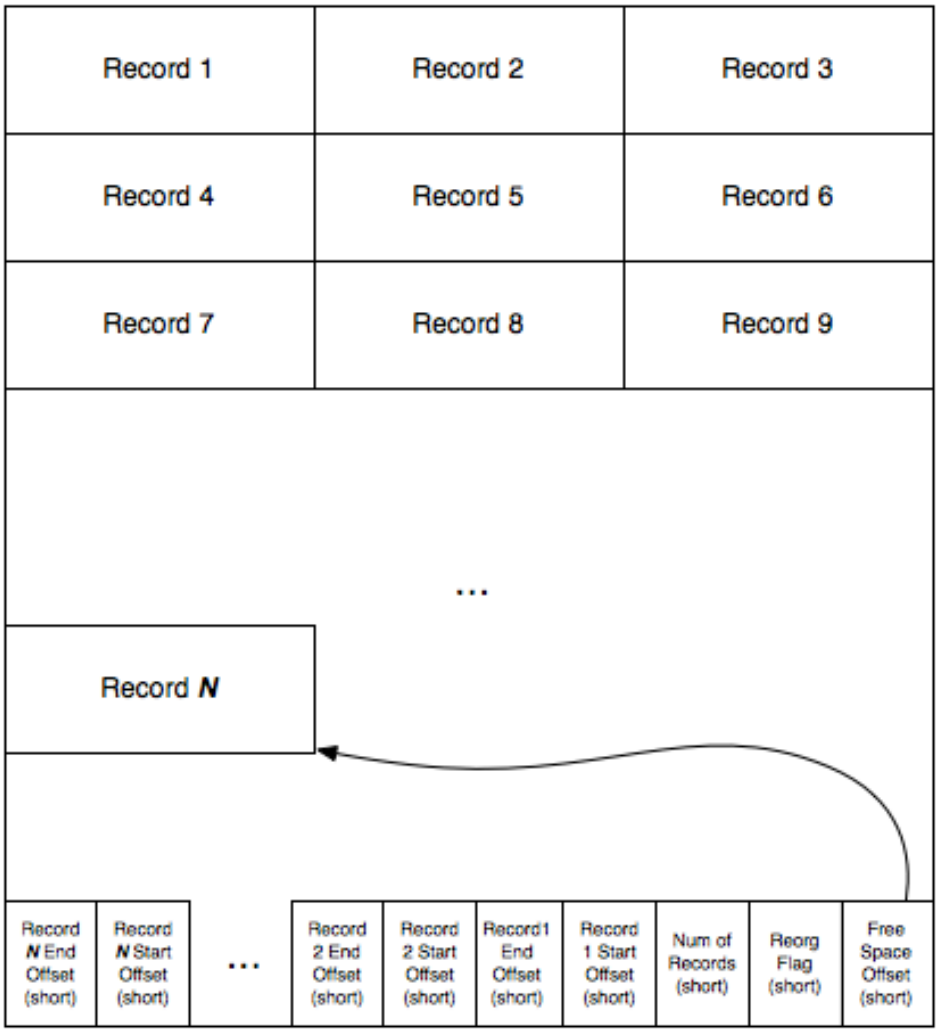
The above image depicts a simple header page layout. Each file that is created will have at least one header page associated with it. The header pages are responsible for keeping track of all the free space associated with the pages in the file.

As shown above, the first 4 bytes of the header page will consist of a value indicating whether another header page exists or not. The idea behind this was, since every page is allocated 4096 bytes of data, at some point a header page may exceed its storage capacity (4096) in the event that a file contains many records. In this situation a value indicating the value of the next header page will be stored in the first header page. If there is only one header page, a negative value will be written to this data area to denote

this (i.e. invalid next header page). This structure provided us with the capability to chain header pages together in the event of large amounts of data written to a file. Additionally, these fields will be used for various methods such as writing data to file. If multiple pages exist, the first header page will be checked to see if there is enough space to write the data out to in one of its page. If not, the next header page will be read and the process repeated.

The value next to the *nextHeaderPage* data is the number of pages that are currently in the header. Next to this value follows the amount of free space that is available in each respective page of this header in order (1,2,3…). This can value can be used to determine whether or not a page has enough free bytes available to write data to when attempting to insert a record. Since each value is a short (2 bytes), these values can be calculated and determined with minimal effort.

Page (non-header) Structure



Page Layout

Liangjun Feng
John Palomo
Group ID: 15

The above picture illustrates our page structure for records. Each page consists of a number of records and variety of fields located in the footer of the page. Records are written in order. At the bottom of each page contains important information regarding important features of the page that will facilitate read/writes. The first field is the free space offset. This field is a short value that indicates the offset bytes where the beginning of the free space in the page exists. The next field, reorg flag, is another short value that is basically a counter. Since we know the number of free bytes that each page contains from the header, we will try to insert records into the first page that contains enough free space. However, the header just contains the free space, and not necessarily the contiguous free space. If we try to insert a record into a page but cannot find enough contiguous free space, we will increment the reorg flag. We will establish a count (e.g. 3), which indicates the number of insert misses at which we must reorganize the records in the page. This will be implemented in part to of the project. The next field is a simple value to keep track of the number of records in the page. Lastly, the next to short values taken together represents the records' begin and end byte offset. We needed to keep track of the begining and end in order to determine the total free space that is available when we delete records.

Page File Manager

In the `PagedFileManager` class, we added a map of strings to unsigned integers, called *fileDirectory*. This map serves as a directory that we query to determine whether or not a file is currently open, and how many file handles are associated with this file if it is open. Every time a file is opened, the file name is added to the directory (if not in the directory) and the file handle count is incremented. When the file is released, the file handle count is decremented and if the count == 0, the file is removed from the map. This was needed to ensure that files are opened and closed properly. If a request is made to destroy a file, we needed a way to ensure that there were no file handles currently associated with the file (i.e. reading/writing data to the file). If a destroy file request is made and a handle count is not 0, the file will not be destroyed.

Record Based File Manager

The majority of the changes for this portion of the project were made in the `RecordBasedFileManager` class. Like the `PagedFileManager`, the `RecordBasedFileManager` contains a directory called the *filePageDirectory,* which holds a map that correlates a filename to a vector of free space values in the pages of that file. This map is populated when reading the free space available for each respective page from the page header of the file. The rest of this write-up will explain the details for many of the implemented methods or created methods that were used to achieve our design.

createFile(const string &fileName)

The create file method simple creates a file with its initial header page. The header values, nextHeaderPage, numberOfPages, and freeSpace of header are set to 0, 1, 0 respectively. Free space is set to zero so that the header page is not considered as a possible page to write records to when calling the insertRecord method.

destroyFile(const string &fileName)

This simple method that destroys a file by the given name if and only if the file does not reside in the *fileDirectory* of the `PagedFileManager` class. It additionally destroys the entry in the *filePageDirectory* associated with the file.

Liangjun Feng
John Palomo
Group ID: 15

openFile(const string &fileName, FileHandle &fileHandle)

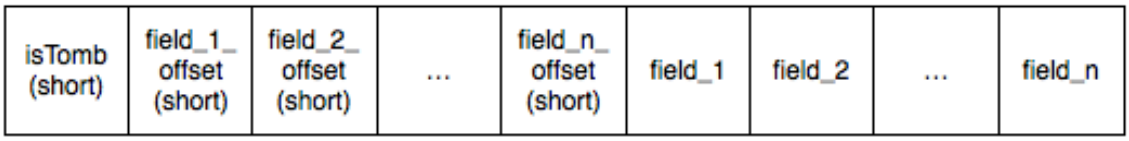
This method opens a file by the given name passed to it. In addition to opening the file, the header of the file is read and the *filePageDirectory* is loaded into the map.

closeFile(FileHandle &fileHandle)

This method will close a file by the given name passed to it. Since the *filePageDirectory* may have been updated since the file was opened, the updated values are written back to disk in the header page to update the free space in each page in the file. The method also releases the handle on the file in the *fileDirectory*.

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

The bulk of our work was done in this method. The method first "encodes" the record with specific headers that allow us to determine the field offsets when reading the records. The following is the format for our record.



Record Layout

1. isTomb indicates whether this record has been moved. If the value is 1, then the next 4 bytes can be interpreted as the new address of the record.
2. Field offset indicates the offset where a particular field ends.

The method first finds a suitable page to write the file to by reading the free space available in the *filePageDirectory*. Once a potential page has been found, the method will then read the page into memory and find either a deleted slot or write the record to the free space. It should be noted that the record would not be written to a deleted spot if the record were less the 1/3 the available space to write to. In this case, the record will be written to the free space. If the record cannot fit into a deleted slot (either too small or too big) and cannot fit into the free space, the reorg flag will be incremented, and a new page will be allocated and the record will be written to that page. When the method finishes, it will update the *filePageDirectory* with the new free space values, which will be written on a call to closeFile.

*appendPageWithOneRecord(FileHandle &fileHandle, const void *data, int dataLength)*

This simple utility method simple creates a new page facilitating the construction of the new page. The method will first prepare the data by encoding the record and then initialize the footers at the bottom of the page with the correct values and then write the data to the page.

*prepareDataForNewPageWrite(const void *record, void *pageData, int dataLength)*

This is a simple utility method that will create the footers of a new page and put the new record date at the start of the page.

Liangjun Feng
John Palomo
Group ID: 15

*readFooter(void *endOfPagePtr, short &reorgFlag, short &freeSpaceOffset, short &numberOfRecords)*
This is another utility method that is used to read the footers of a page, putting the values of the page footers in their respective parameters locations.

*readRecord(FileHandle &fileHandle, const vector<Attribute> &recordDescriptor, const RID &rid, void *data)*
This method reads a record given its record id. It performs this action by first reading the page where the record is located in memory, then reading the beginning and end address of the slot number. Once the beginning and end address are deciphered, the record is then found in the page and read out.

*getRecordLength(const vector<Attribute> &recordDescriptor, const void *data)*
This method returns the total byte length of the record including the additional information needed to decode the record (i.e. isTomb, field offset1, field offset2, etc)
*readHeaderPage(FileHandle &fileHandle, int &currentHeaderPage, vector<short> * spaceLeft)*
This method will read a given header page in the *filePageDirectory* to be able to determine the amount of free space is available in the pages contained in the header.

*writeHeaderPage(FileHandle &fileHandle, int currentHeaderPage, int &nextHeaderPage, vector<short> * spaceLeft, unsigned &currentPage)*
This method will write out a header page to disk. Additionally, if there is not enough room in the header page to write all the pages, an additional header page will be allocated, the previous header pages *nextHeaderPage* value will be updated and all header pages will be written out to disk. Each header page can store 2040 entries, or pages before requiring an additional header page.