CS222 Principles of Data Management: Project 1

Contents

[Overview 2](#_Toc401001039)

[Design 2](#_Toc401001040)

[Paged File Layout 2](#_Toc401001041)

[PagedFileManager Class 2](#_Toc401001042)

[FileHandle Class 3](#_Toc401001043)

[RecordBasedFileManager Class 3](#_Toc401001044)

[Implementation 4](#_Toc401001045)

[PagedFileManager Class 4](#_Toc401001046)

[FileHandle Class 4](#_Toc401001047)

[RecordBasedFileManager Class 4](#_Toc401001048)

[Helper Class: SlotDirectory 5](#_Toc401001049)

[Helper Function: get\_free\_index\_and\_entries() 5](#_Toc401001050)

[Helper Function: write\_default\_index\_bytes() 5](#_Toc401001051)

Jian Luan

ID: 76111817

# Overview

The first project for CS222 is to implement the very basic level of a relational database, which includes the paged file manager, file handle class, and record based file manager. The page file manager handles the creation, destruction, and access to the underlying page file. The file handle operates on the page file created by the page file manager to blindly read/write pages into the paged file. Finally, the record based file manager is the highest level manager which reads and writes records into the paged file.

# Design

Much of the design of the low level database management classes follow canonical designs from the text book. For the classes RecordBasedFileManager, FileHandle, and PagedFileManager, skeleton code was provided to develop from which encompassed much of the design required for this project. However, many smaller design decisions were made in order to facilitate a robust paged file manager which can handle various types of input quickly.

## Paged File Layout

Each paged file created by the PagedFileManager has a unique header appended to be beginning of the file, before actual pages begin. This header is checked by the PagedFileManager to make sure that the file being opened is indeed created by the PagedFileManager.

Each page contains 4096 bytes of data. Within each page, records are stored from the beginning of the page. Each record also contains a record header of unsigned int size which indicates the length of the record. This was appended to the record to ensure reading of a proper length of the record, especially in situations where a record is deleted, and a shorter record was written over this space, not completely filling the void where the previous record existed.

A slot directory is also written to the end of each page. This directory indicates the first free space position offset from the beginning of the page, the number of records on the page, and the position of the start of each record. Using this directory, the RecordBasedFileManger can quickly access any record on a page without having to sequentially scan through the page until it reaches the record. Having the page slot directory at the end of each page sacrifices some storage space to gain random access capabilities. The slot directory is written backwards from the end of the page forward in the sequence [free space offset][number of slots on page][first slot offset][second slot offset]… and so on.

## PagedFileManager Class

This class is designed to handle low level operation which create/destroy files, but cannot alter the data within. This class is designed using the singleton pattern because we only want a single instance of the PagedFileManager to exist across the entire database. Having multiple instances can cause various unexpected problems when accessing paged files.

## FileHandle Class

The FileHandle class is designed to handle single paged files. Each instance of the FileHandle class can handle up to one paged file and operate on this file. This means that the class needed to be designed in such a way that FileHandle can be aware of the input file which it operates on. Hence, functions such as a custom constructor and loadFile() member functions were created.

## RecordBasedFileManager Class

The RecordBasedFileManager class reads and writes data into paged files and controls the exact location of each data read/write. This class needs to be aware of the structure of the page and the layout of the memory within each page in order to correctly retrieve/write data. RecordBasedFileManager utilizes FileHandle class and PagedFileManager class in order to fully control the reading and writing of data into paged files as well as paged file creation and deletion.

# Implementation

## PagedFileManager Class

In order to ensure that each page file opened was actually created by the PagedFileManager, a small 17 byte token is inserted into the beginning of the page on createFile() in order to make the page recognizable to the PagedFileManager. This token is defined as a static const string in order to prevent the data from ever changing. Thus, when the PagedFileManager’s openFile() function attempts to open a file which was not created by the class, then a -1 return code is passed. If the PagedFileManager attempts to create file with a file name that already exists, then the class will also return an error code of -1.

## FileHandle Class

For the FileHandle class, a paged file’s name must be tied to the file handle for the file handle to operate. Thus, a loadFile() function was created to open the file for both reading and writing in binary mode. Here, C++ fstream is used for its ability to automatically close the file when leaving scope. Various helpers also exist to facilitate proper checking of FileHandle class states such as isDbFile() and isUsed(). These member functions return whether the file was created by the PagedFileManger and checks if the file is open respectively. Member function getNumberOfPages() accounts for the offset of the token which was written at the beginning of the paged file for it’s calculation of number of pages. Thus, number of pages is calculated as:

This accounts for the offset of each page as well as the token size.

## RecordBasedFileManager Class

This class is the most involved class implementation-wise. Member functions createFile(), destroyFile(), openFile(), and closeFile() all call upon implementations from the PagedFileManager class under the hood.

Member function insertRecord () handles the writing of a record into the paged file at the first available free space position which is large enough to contain the record. In order to do so, the true length of the record must be evaluated using both the record descriptor and the binary data passed in. Both TypeReal and TypeInt data from the descriptor can be parsed easily using a fixed size. However, TypeVarChar is more involved due to the fact that TypeVarChar record descriptor supplies a max length instead of actual length. The actual length of a TypeVarChar record must be parsed out of the binary data. To simply this process, a function compute\_descriptor\_size() was created to return the true size of the record to be written. Next, free space is checked and a new page is inserted if necessary. The page which the new record will be written to is then loaded and its slot directory is read. From here the record is written at the correct position from the beginning of the page and the slot directory is updated and rewritten. Finally, the RID data is passed back from the function.

Member function readRecord() retrieves a record based on RID and FileHandle objects. Based on the RID, the RecordBasedFileManager will load the page number desired and read the slot directory to find the position on the page of the desired slot number. Because there is a prepended record length header at the beginning of each record, we can figure out how long the record is based on this header, without needing to use the record descriptor. The readRecord() member function then reads a record length number of bytes from a specific offset to retrieve the record.

Member function printRecord() is a debug utility function which parses a set of binary data based on a record descriptor. The parsing scheme is the same as outlined in insertRecord().

## Helper Class: SlotDirectory

In order to abide by object oriented design, a SlotDirectory class was created to ingest page slot directories to efficiently look up the correct indices of each record on the page. This class has specific functions for reading either, only the slot size and free space position, or reading the entire directory. This separation of functionality allows quicker access to free space estimation and scans, in which case we don’t need to know exactly how a page is laid out. This class also contains convenience member functions which return the position of the next free slot number, or the offset for a slot number. SlotDirectory also handles the writing of new slot directory information into the current slot directory.

## Helper Function: get\_free\_index\_and\_entries()

This helper function allows us to quickly access only the last two unsigned int sized values on a page without having to read and parse the rest of the slot directory, adding a small performance improvement.

## Helper Function: write\_default\_index\_bytes()

This helper function writes a default slot directory on a page, useful for being called together with appendPage() function. It writes two values to the end of the page: an offset for free space of 0, and a slot directory size of 0.