Project 1 - Design and Implementation Summary

CS222: Principles of Data Management

Christopher A. Wood
Tamir Husain
January 24, 2014

Task

Write a report to briefly describe the implementation of your paged file and record-based file systems.

Summary

In this document we outline the design and implementation of our paged file and record-based file systems. We describe the memory layout on disk, the bookkeeping methods used to maintain data correctness and integrity, and also describe additional utilities that were developed to support all required functionality.

Persistent Storage Management and Memory Layout

As per the project specification, all data written to the database is saved on the machine's hard disk after every write operation. Buffer management and physical page writes to database files, which are actually files on the underlying file system, are delegated to the operating system. The lowest layer of our DBMS, the `PagedFileManager`, is simply responsible for specifying the offset in these files where page-sized chunks of bytes are read and written. Higher levels of the DBMS interact with pages through a `FileHandle` instance, which is allocated and populated through the `PagedFileManager`. In order to properly support the `openFile`, `closeFile`, and `destroyFile` methods in the `PagedFileManager` class, a collection of reference counts, each of which corresponds to the number of open file handles associated with a particular database file, is maintained by the `PagedFileManager`. Internally, this is implemented as a map data structure with file name string keys and integer reference count pairs. The `openFile` method will increment existing reference counts or insert new file names into this map upon invocation, the `closeFile` method will always decrement the reference count for the file to be closed (if it exists in the table), and `destroyFile` will always check to ensure that no open file handles refer to a particular file prior to removing it from the file system.

Each `FileHandle` instance used to interact with pages in a single database file maintains only the number of pages currently allocated to the file as an instance variable. However, in order to ensure that this number is correct (i.e., it accurately reflects the number of pages on disk even if another file handle for the same database file is used to append a page), the operating system is always queried to update this variable when the `readPage`, `writePage`, `appendPage`, and

`getNumberOfPages` methods are invoked. If this procedure was not done, multiple file handles referring to the same database file could end up in an inconsistent state if used improperly. Beyond this simple bookkeeping, all `FileHandle` operations are simply wrappers for underlying file system I/O routines, supporting the movement of page-sized chunks between main memory and the hard disk.

The top-most layer in the DBMS, the `RecordBasedFileManager`, is where the majority of the logic for the file and record management lies. We will describe the implementation techniques for record operations in the subsequent section of this document. In order to maintain information about the database file, such as the number of pages available, how much free space is on each page, etc., the `RecordBasedFileManager` uses the linked-list heap file organization structure outlined in [1]. Specifically, page 0 of the database file is reserved for the header index which maintains pointers to the start of a collection of linked lists corresponding to pages in the same “free space class,” which are simply pages that have a garunteed lower bound on the amount of free space available for record insertion, as well as the total number of pages in the file (excluding the header page) and various metadata used for header file verification. Figure 1 below illustrates a sample instance of our database file storage on disk; metadata is not shown for brevity.

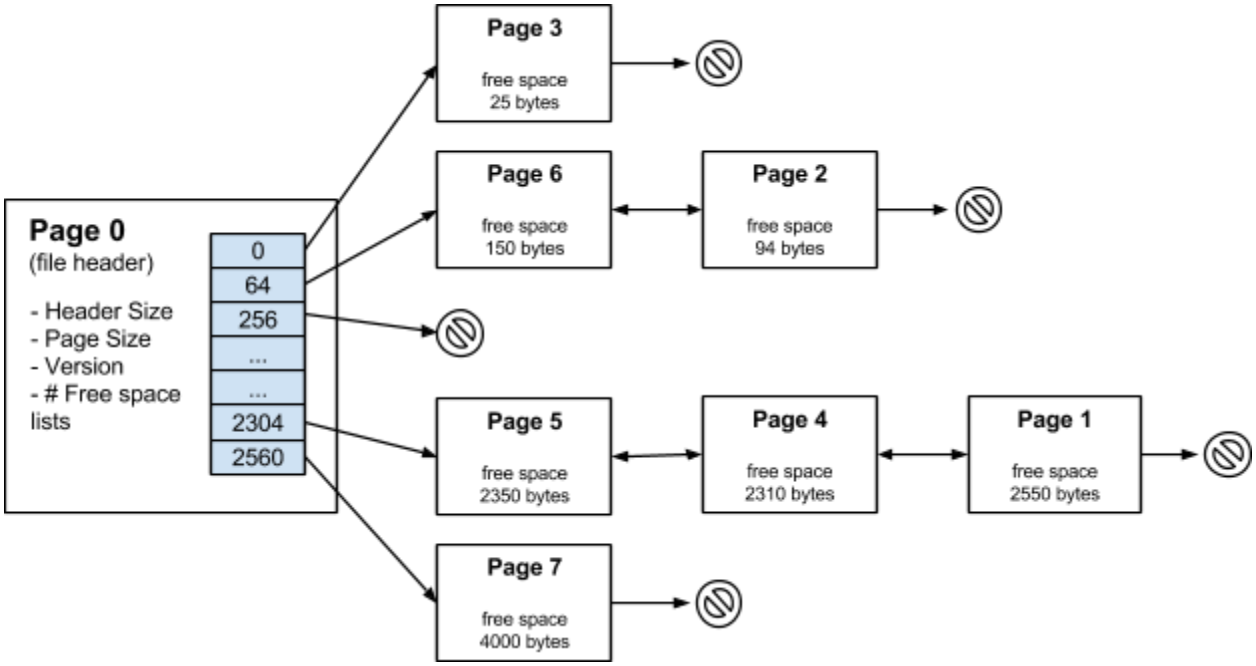

Figure 1. An example of a linked-list heap file organization instance.

On top of the heap file organization, each page has a uniform organizational structure to support variable length records. Following the design presented in [1], the end of a page is reserved for a page index header, which contains information used to traverse the free space linked list of pages, the page number, number of slots (records) N on the page, and an offset from the base page address to the first available memory slot. If N slots exist on a particular page, then N slot

index entries are prepended to this page header index, where each entry is a tuple storing the size of the total record and corresponding offset from the base page address where it is located. Records are stored in contiguous order on disk and also packed as closely as possible to avoid internal fragmentation. Also following the design strategy presented in [1], variable length records are supported by storing each record as the concatenation of an array of field offsets and the actual field data. More specifically, the array of offsets is used to determine the offset from the base record address where the field begins, and the length of the field is equal to the difference between subsequent field offset and the current field offset. The page and record layout used to support variable length records are illustrated together in Figure 2 below.

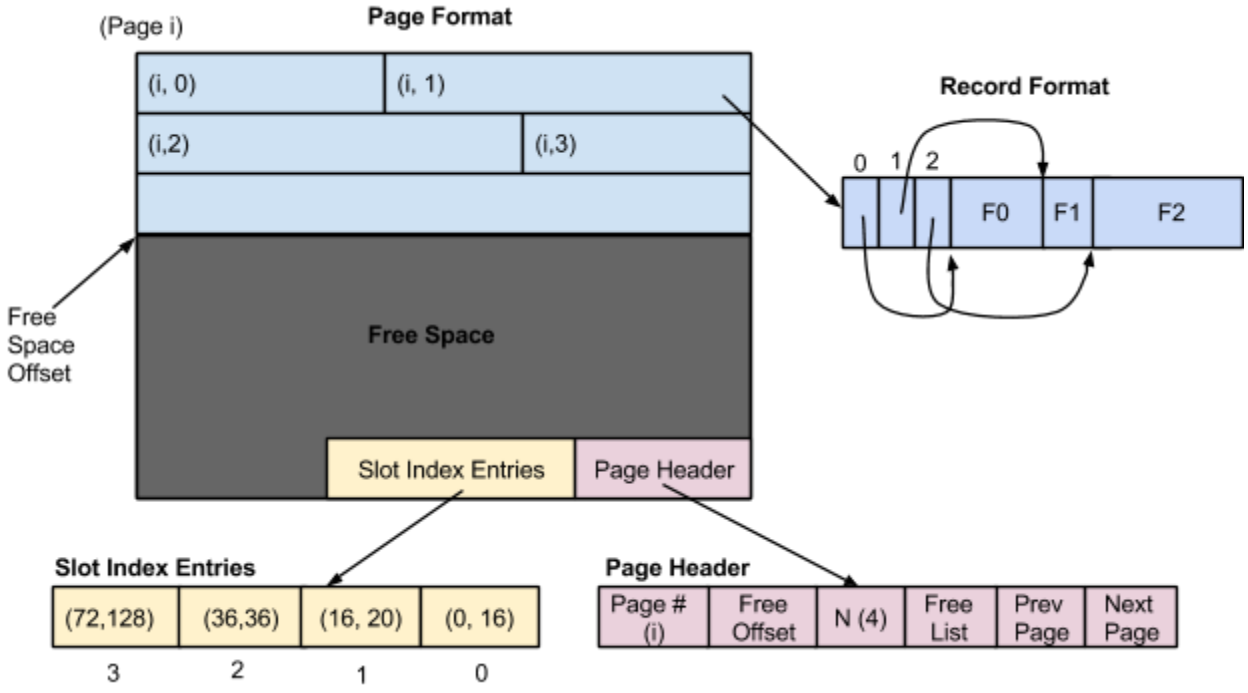

Figure 2. An example instance of a page layout on disk, illustrating the contents of the page index hader, slot index entries, and the format of each record.

Record Based File System API Implementation

The `RecordBasedFileManager` handles all page read/write operations necessary to fulfil user requests, such as inserting or reading a particular record. In order to insert a record, we must first determine the length of the record. We do this by iterating through the provided `recordDescriptor` and incrementing the size of the record based on each element. Integer and real values are fixed size, and we use the prepended size of the string in order to not waste space allocating the maximum size of the variable length field. Once we have found the size the record will take on disk, we query the `RecordBasedFileManager` to `findFreeSpace`. This function will determine if an existing page has enough free space to hold this new record or append a new page if none exist. This lookup iterates through our freespace lists, checking for existance of a list with enough free space. At every free space list index, if there are any pages

there, we are guaranteed they have at least a given amount of free space, so we do not need to traverse the entire list. We have only 11 free space lists, and the list head data is in the file header, so this operation is $O(1)$ only reading a single page into memory.

On finding a page with enough available space, we write the record itself after the previous last record, and we update the page header and new slot index entry. After writing the record, we decrement the amount of free space available for this particular page. If this requires invalidates the guarantee of free space availability on its current list, it is removed and placed at the head of a list where the guarantee still holds true. These operations will only ever occur at the head of the lists, so the cost is still $O(1)$, at most reading/writing three pages (header, current page, and page on new list with which we swap). After this is done, we simply update the passed in RID with the new page number and slot number and then update the file header on disk. Upon success, the record is successfully inserted and given to the OS filesystem to be written to disk.

A `readRecord` operation is even simpler than the `insertRecord` operation. We can immediately pull the correct page into memory, as the page number is given to us in the RID. From here, we read the location in memory to which the slot index points. This gives us the location in memory where the record exists so we can extract it and subsequently copy the data into the user's buffer. Clearly, the overall complexity of this operation is $O(1)$.

Additional Utilities

To facilitate the development of this part of the project, we also abstracted the set of possible return codes to an enumerated type defined in the `returncodes.h` file. The supporting `returncodes.cpp` is used to map enumerated values to human-readable strings, making the identification of errors quite simple. Furthermore, we developed a singleton logging class that can be used to write any debug message to stderr. Debug and error messages are toggled by flipping the `DEBUG_ENABLED` macros in `dbgout.h`.

References

[1] *Database Management Systems*, 3rd edition, by R. Ramakrishnan and J. Gehrke, McGraw Hill, 2003.