Project Stage 1.0: Part 3: Heap Files

Due date: Wednesady, May 11, 11:59pm.

Heap File Layer

Heap files are the core of DBMS data storage, and FLOPPY is no different.

FLOPPY Heap files. A FLOPPY heap file is a `tinyFS` disk file that consists of a single file header page (`tinyFS` Page 0) and a sequence of data pages.

At present, we proceed under the assumption that **all** data pages in a **single** heap file store exactly the same records and are organized in exactly the same way. The heap files shall store only **fixed-length records.**

Diversity. Different FLOPPY heap files may (and will) store differently structured records. Record structures for each of FLOPPY heap files are determined at **run-time** – after FLOPPY processes the appropriate `CREATE TABLE` command. The structure of the record for each FLOPPY heap file must be decided during the `CREATE TABLE` processing stage, and it must be stored on the file header page.

File Header Page. There should be enough free space on the file header page of each heap file to store all information you need. At a minumum, the following information must be present on each header page:

1. Table name.
2. Record description. This includes the order of the fields, field sizes, and any record header information you need (e.g., the tombstone byte, or a timestamp).
3. Record size. The length of the record in bytes must be stored.
4. PageList. The order of pages in the heap file needs to be controlled by the DBMS — new pages may be inserted at random locations in the heap file. The starting point of the PageList list shall be located on the file header page.

5. FreeSpace list. The DBMS also must keep track of locations in the heap file where additional tuples can be inserted. This is known as the FreeSpace list. The pointer to the first such page in the heap file shall be located on the file header page.

You can allocate space and store any other information on the file header page, that you believe will help FLOPPY in data processing tasks.

Data Page structure. The data page of a FLOPPY heap file shall consist of a Page Header and the record storage space, with some leftover space possibly available at the end of the page (the leftover space is less in size than a single record to be stored in the data file).

Page Header. The page header of the FLOPPY heap file data page shall be located at the top of the page (starting a the byte offset that does not contain any `tinyFS`-specific information). The size of the page header must be a constant that you preset ahead of time as part of your design process. That is: the size of the page header for *any* FLOPPY heap file data page shall always be the same, regardless of the nature of the specific table stored in the file.

The structure of the page header shall also be fixed. The page header shall contain, at a minimum, the following information:

- Filename, PageId. (this you might get for free from `tinyFS`, but make sure that you can authenticate any disk page as belonging to a specific file and having a specific unique id within the file).
- Record-tracking fields. Your page header must contain information about the records stored on the page. Depending of the specific storage approach you take, you may be able to get away with storing different types of data, but some of the things to store may include:
  - Record size.
  - Record structure. Might not be needed (it is avaialble from the file header), but might make you life easier.
  - Number of records on the page. The total number of records that can be stored on the disk page.
  - Number of occupied/free slots. Number of records slots that contain/do not contain a record. You can store one of these.
  - Disk page occupation bitmap. A bitmap showing which slots on the page are occupied and which are empty. This is useful if you are not compressing the content of the data page on deletion.
  - Access timestamps. A few timestamps, showing page creation time, last time the page was flushed to disk, and, potentially, other useful timestamp information.
  - NextPage pointer. Your page must have a pointer to the next page in the disk file.

– FreeSpace list pointer. You page must have a disk address slot to store a pointer to the next page in the FreeSpace list.
– Back pointers. For every linked list you are maintaining, you may consider storing the back links as well to make the lists double-linked.

In addition to this information, **you can store any other data** on the header page. Please plan for the size of the header page that can accommodate all necessary information.

Records. You can to select the actual structure of your records, and whether or not *any* record headers are necessary. When doing so, please take into account the following:

- We are enforcing the *integers must start at offsets divisible by 4, floats must start at offsets divisible by 8* rule for all records. The values of `DATETIME` type may need to be treated as `INT`s or `FLOAT`s depending on how you represent them (if you represent them as strings, do not worry). You can choose whether to use padding, reordering, or a combination of thereof to properly structure your records.
- Record structure shall be encodable in a straightforward way for a representation that is to be stored in on the file header pages.

APIs

You have to implement a variety of functions for accessing data on `FLOPPY heap files`. We break it down to file creation, file header functionality, data page functionality, record-level functionality and CRUD functionality. The APIs below are approximate. They identify the names and the purposes of the functions, but the actual list of function arguments is something you may need to finalize depending on your application.

Note: Chances are, you need a `Buffer * buf` parameter for each function. I am going to ignore it in what follows.

File creation. The following data file-level functions shall be implemented.

1. `int createHeapFile(char * tableName, char * createTable, int volatileFlag);` Create a file name based on the name of the table and the text of the `CREATE TABLE` statement. Alternatively, you can replace the `char *createTable` argument with a `struct` that describes the structure of the table based on the `CREATE TABLE` statement (to move parsing outside of this layer). The `int volatileFlag` parameter indicates whether the table shall be stored in persistent or volatile storage.
2. `int deleteHeapFile(char * tableName);` Delete a heap file from disk, remove all its pages from the buffer.

File Header. For the file header, we largely want to encapsulate access to the information stored in the file header. Some examples are:

1. int heapHeaderGetTableName(FileDescriptor fileId, char * name); Get the table name from a given header page.
2. int heapHeaderGetRecordDesc(FileDescriptor fileId, struct <whatever> * recordDesc); Get the file descriptor structure. The second attribute can be replaced with a char * (i.e. byte array).
3. int heapHeaderGetNextPage(FileDescriptor fileId, DiskAddress * page); return the address of the next page in the PageList list.
4. int heapHeaderGetFreeSpace(FileDescriptor fileId, DiskAddress * page); return the address of the next page in the FreeSpace list.

For each of these functions, as appropriate setter function shall also be implemented.

Data Page. There are two key operations on the actual data on the data page: putRecord() and getRecord(). Additionally, work with the data page header should be encapsulated the same way you encapsulate the file header management.

1. int getRecord(DiskAddress page, int recordId, char * bytes); given a page id and a location on the page (represented as a record id), retrieve the contents of the record into the output byte array.
2. int putRecord(DiskAddress page, int recordId, char * bytes); given a page id and a location on the page, place the given record onto the appropriate slot of the page.

Notes: There are other possible attributes to these functions. It may be easier to use an actual pointer to the beginning of the page instead of the DiskAddress as the input parameter. You may want to pass the length of the record to this function (otherwise, the function can determine the length of the record by grabbing the value from the page header). You may also want to replace the recordId parameter with and offset parameter, which points at the actual byte offset for the location where the record is to be placed. (Alternatively, you can implement multiple versions of the getRecord() and putRecord() based on what information you can pass into the function.

1. int pHGetRecSize(DiskAddress page) or pHGetRecSize(char * page). Obtain the size of the record from the page header.
2. int pHGetMaxRecords(DiskAddress page) or pHGetMaxRecords(char * page). Obtain the number of record slots on the page from the page header.

3. int pHGetNumRecords(DiskAddress page) or pHGetNumRecords(char * page). Obtain the current number of occupied slots on the page.

4. int pHGetBitmap(DiskAddress page) or pHGetNumRecords(char * page). Obtain the page occupancy bitmap. (If your page header stores the bitmap).

5. int pHGetNextPage(DiskAddress page, DiskAddress * nextPage) or pHGetNextPage(chair * page, char * nextPage): given a page, return the next page in the file order.

6. int pHGetNextFree(DiskAddress page, DiskAddress * nextPage) or pHGetNextFree(char * page, char * nextPage): given a page, return the next page with free elements in it (if possible).

You will need to add an appropriate get function for each element of the data page header that you introduce.

You also will need to implement the complimentary set functions for each of the get functions above. Some of the more interesting setter functions are:

1. int pHSetBitmapTrue(<dataType> page, int index); this function sets the position number index of the data page header occupancy bitmap to True (data page record slot number index is occupied).

2. int pHSetBitmapFalse(<dataType> page, int index); this function sets the position number index of the data page header occupancy bitmap to False (data page record slot number index is not occupied).

3. int pHDecrementNumRecords(<dataType> page) and pHIncrementNumRecords(<dataType> page): decrement/increment the number of occupied record slots on the data page by one.

Record-level data access. These functions take as input a byte array specifying with the content of a record, and read/write specific portions of the record. You can either pass a record descriptor (we use the data type RecordDescriptor here to indicate values that represent the record structure), or you can query the record descriptor directly from the access functions.

1. int getField(char* fieldName, char * record, RecordDescriptor rd, char * out); given a byte array storing a record, and a field name, retrieve the value of the field.

2. int setField(char * fieldName, char * record, RecordDescriptor rd, char * value); given a byte array storing a record, and a field name, overwrite the value of the field with a given value.

3. int getRecordHeader(char * record, RecordDescriptor rd, char * out); retrieve the header of the input record.

In addition, you can implement a get and a set function for each of the values stored in the record header.

Heap File CRUD operations.

There are three major functions at this layer:

1. `int insertRecord(char * tableName, char * record, DiskAddress * location);` given a name of the table, and a record, insert the record into the data file (optionally, return the disk address of the page on which the record is stored).

2. `int deleteRecord(DiskAddress page, int recordId)` or `int deleteRecord(char * page, int recordId)`: given a location of a record on disk, delete the record.

3. `int updateRecord(DiskAddress page, int recordId, char * record)` or `int updateRecord(char * page, int recordId, char * record)`: replace the data at a given address (disk page, record id) with the new `record`.

Please note, the `insertRecord()` function can be directly called in response to the `INSERT INTO Table VALUES()` SQL command. At the same time, `updateRecord()` and `delateRecord()` require extra work before the can be called during the execution of `DELETE FROM` or `UPDATE` commands.

At this point, `insertRecord()` only inserts the record onto a disk page. At a later stage, we will add index management to this function.

Coming Up Next

The next specification, to come out on Wednesday, *May 4* will detail the `sequential file` API that you must implement.

We will also try to release the API specs for the `sequential files/B+trees` and `hashed files` no later than by Wednesday, *May 4*.

Good Luck!