Project Stage 1.0: Part 2: Read() and Write() functions

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

Read() and Write()

This document describes one of the part of Stage 1 of the project: the read and write functions connecting the Buffer Manager to the remainder of the FLOPPY infrastructure.

Read/Write Operations

The FLOPPY Buffer Manager is responsible for making sure that the disk pages that the upper layers of FLOPPY need to read from/write to are in the FLOPPY buffer when they need to be. However, the `readPage()` and `writePage()` functions you had to implement for the Buffer Manager layer, by themselves, do not actually read any data, or change any data on the disk pages.

The honor to do so falls on the functions described in this document.

The Read/Write layer of FLOPPY is the key communication channel between the disk access layers, largely concerned with managing the disk I/O operations, and the implementations of the relational algebra queries and CRUD operations, largely concerned with the actual data stored on the disk pages.

The read/write functions you have to implement in this layer are agnostic of the nature of the data being read/written. This allows them to work well with any page residing in either the persistent or the volatile part of the buffer.

For the Read/Write layer of FLOPPY you will implement the following functions:

function	Purpose
`char * getPage(Buffer * buf, DiskAddress page);`	return the byte array with the contents of entire page
`int putPage(Buffer buf, DiskAddress page, char * data);`	make `data` the contents of a disk page
`char * read(Buffer * buf, DiskAddress page, int startOffset, int nBytes);`	return a portion of the disk page
`int write(Buffer * buf, DiskAddress page, int startOffset, int nBytes, char * data);`	write data to disk page
`char * readVolatile(Buffer * buf, DiskAddress page, int startOffset, int nBytes);`	read from volatile storage
`int writeVolatile(Buffer* buf, DiskAddress page, int startOffset, int nBytes, char * data);`	write to volatile storage

Brief descriptions of these functions follow.

`char * getPage(Buffer * buf, DiskAddress page).` This function takes as input the `Buffer` structure and the disk address (`page`) representing the page, and returns back the byte array representing the entire contents of the disk page.

If the disk page does not exist, return `NULL`.

Please also note that `getPage()` is a *universal* function: it must successfully read from both persistent and volatile buffer storage. The function shall determine whether the disk page is located in the volatile storage or the persistent storage in the buffer, and shall perform a successful read from either location.

`int putPage(Buffer * buf, DiskAddress page, char * data).` This function, given a buffer structure `buf`, a disk address `page` and a byte array `data`, replaces the contents of the disk page `page` with the contents of the `data` array.

Notes: Some things to consider:

- `data` must be at least the size of a single disk page. If it is less than a single disk page in size, return an erro code. If it is larger than a single disk page size N, use the first N bytes from `data` as the news contents of the disk page `page`.
- If `page` does not exist, create it.

Please also note that `read()` is a *universal* function: it must successfully place the page into persistent or volatile buffer storage depending on the file into which the page is being placed. The function shall determine whether the disk page is located in the volatile storage or the persistent storage in the buffer, and shall perform a successful write into either location.

`char * read(Buffer * buf, DiskAddress page, int startOffset, int nBytes).` This function takes as input the buffer structure `buf`, an address of a disk page `page`, a starting position `startOffset` on this page and the number of bytes `nBytes` to read. It returns back the contents of the `nBytes` of consecutive bytes from the given disk page starting at position `startOffset` (including the byte at that position).

Notes: This function requires some care. Please be aware of the following edge cases:

- `page` does not exist. The function shall return `NULL` in this case.
- `startOffset`$-1+$`nBytes`$>$ size of the disk page. Return the byte array containing the contents of the disk page starting at the `startOffset` and ending with the last byte of the disk page.

- `nBytes`> page size. In this specific case, you can return `NULL`, because this is a clear error of judgement.

Please also note that `read()` is a *universal* function: it must successfully read from both persistent and volatile buffer storage. The function shall determine whether the disk page is located in the volatile storage or the persistent storage in the buffer, and shall perform a successful read from either location.

`int write(Buffer * buf, DiskAddress page, int startOffset, int nBytes, char * data).` Given a buffer structure `buf`, a disk page address `page`, a position on the disk page `startOffset`, the number of bytes to write `nBytes` and the byte array of data `data`, this function places `data` into the `nBytes` of storage starting at offset `startOffset` onto the page `page`.

The function returns an success flag/error message code.

Notes: There are some edge cases to consider.

- `data` is longer than `nBytes`. In this case, use only the first `nBytes` from the `data` byte array.
- `data` is shorter than `nBytes`. In this case, fill as much of the disk page with the bytes from `data`, and pad the remaining bytes with `0` character.
- `startOffset`$-1+$`nBytes`> page size. Return error code.

Please also note that `write()` is a *universal* function: it must successfully write to both persistent and volatile buffer storage. The function shall determine whether the disk page is located in the volatile storage or the persistent storage in the buffer, and shall perform a successful write to either location.

`char * readVolatile(Buffer * buf, DiskAddress page, int startOffset, int nBytes).` This function behaves similarly to `read()` except it reads information only from the volatile storage. If the requested disk page is not part of the volatile storage in the buffer, the function returns `NULL`.

`int writeVolatile(Buffer buf, DiskAddress page, int startOffset, int nBytes, char * data).` This is the volatile storage-only version of `write()`. If the page to be written to is not in the volatile storage, the function shall return `NULL`.

In addition to the functions above, feel free to implement any other helper functions. In particular, you may want, for the sake of symmetry, to implement `readP()` and `writeP()` functions: the persistent-storage-only versions of `read()` and `write()`.

Implementation Notes

You are going to build these functions on top of the Buffer Manager layer, which means that all your read/write functions shall make proper use of `readPage()`, `writePage()` and, if necessary, other functions from the Buffer Manager layer.

You cannot encapsulate your Buffer access to the Buffer Manager layer completely though — no Buffer Manager layer function directly touches the data, so upon running `readPage()` you must access the contents of the page directly in the `Buffer` struct object passed to your function.

You can implement `getPage()` and `putPage()` as special cases of `read()` and `write()`. Alternatively, you can use `getPage()` and `putPage()` as you encapsulation of buffer content access, and then build `read()` and `write()` off of it. The specific choice is left to individual teams.

Coming Up Next

The next specification, to come out on Monday, *May 2* will detail the `heap 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!