


Lab 1: Buffer Manager and Heap File

[New Attempt](#)

Due Feb 12 by 11:59pm **Points** 100 **Submitting** a text entry box **Available** after Jan 18 at 12am

Getting Started

This lab will be built upon your code for lab 0. We provide a tarball with some new starter code which you need to merge with your lab0.

1. Make sure you commit every changes to Git and `git status` is clean
2. Download [lab1.tar.xz \(https://psu.instructure.com/courses/2240486/files/144834024?wrap=1\)](https://psu.instructure.com/courses/2240486/files/144834024?wrap=1)  [\(https://psu.instructure.com/courses/2240486/files/144834024/download?download_frd=1\)](https://psu.instructure.com/courses/2240486/files/144834024/download?download_frd=1) and untar it inside your repo. You should now have a directory called lab1 inside your repo
3. `cd lab1` and `./import_supplemental_files.sh`
4. `cd path_to_your_repo` and `git commit -m "Start lab1"`

The code should build without compilation errors. Most of the additional tests are likely to fail. You may list all the tests in your build directory using the `ctest -N` command.

A few useful hints:

1. We *strongly encourage* you to read through the entire document to get a good sense of the high-level design of the storage layer in TacoDB before your start writing code. You may also find it useful to review the lecture slides for the database storage layer (will be released after we finish talking about it). The lab requires you to write a fair amount of code, so **START EARLY**.
2. The description below is meant to be a brief overview of the tasks. **Always** refer to the special document blocks in the header and source files for a more detailed specification of how the functions should behave.
3. You may add any definition/declaration to the classes or functions you are implementing, but do not change the signature of any public functions. Also, do not edit those source files not listed in *source files* to modify below, as they will be replaced during tests.
4. You may ignore any function/class that's related to multi-threading and concurrency control since we are building a single-threaded DBMS this semester. Thus, your implementation doesn't have to be thread-safe.

5. We provide a list of tasks throughout this document, which is a suggestion of the order of implementation such that your code may pass a few more tests after you complete a task. It, however, is not the only possible order, and you may find an alternative order that makes more sense to you. It is also possible that your code still could not pass certain tests due to some unexpected dependencies on later tasks, in which case you might want to refer to the test code implementation in the `test/` directory.
6. You are going to submit not only your implementation but also a write-up for code reading for the `Schema` implementation. Make sure you read through it and understand how a record is laid out in the binary format.

Overview

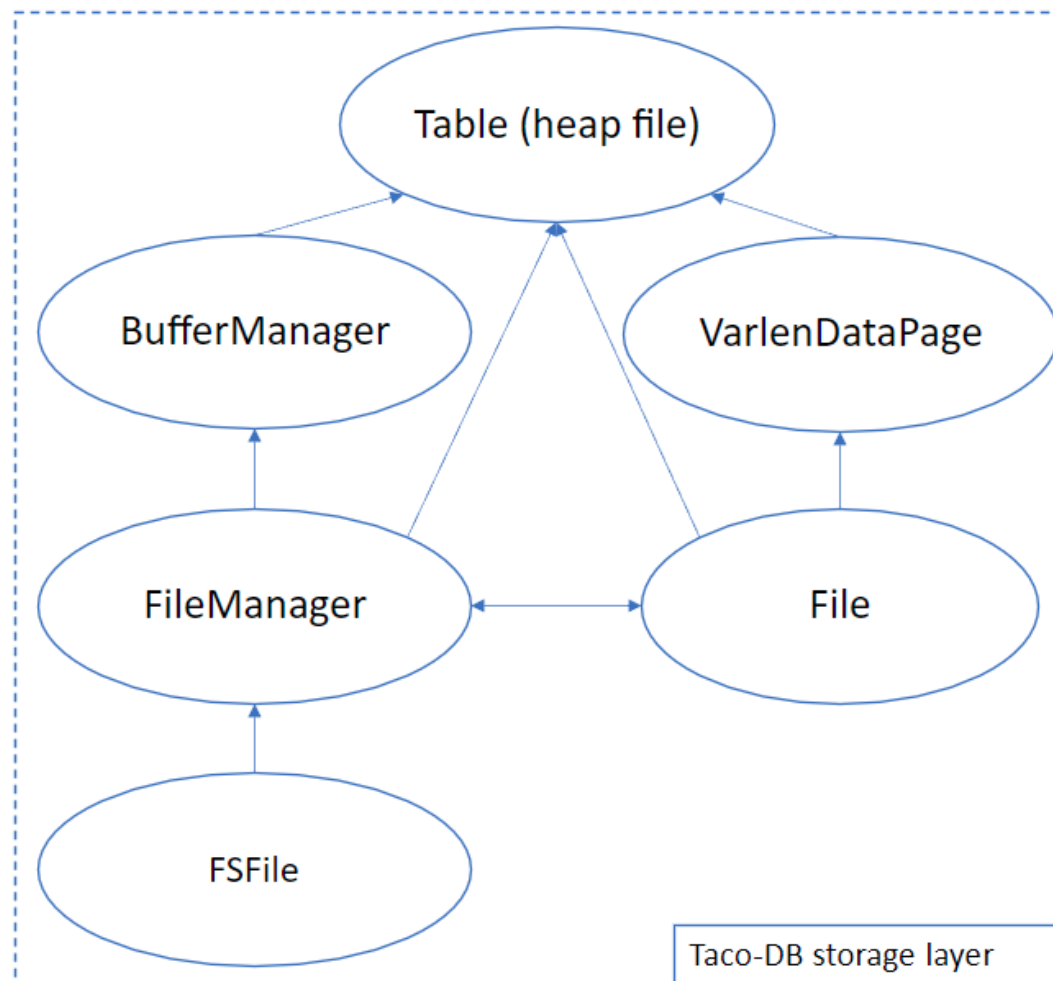


Figure 1: Taco-DB Storage Layer Overview

In this lab, you will work on the storage layer of Taco-DB (see Figure 1). More specifically, the goal is to implement the buffer manager, the data page, and the heap file on top of a middle layer file manager that we provide.

File Manager

Source files to modify: none. You don't have to implement anything in the file manager and the following is only provided as an overview that might be useful to you.

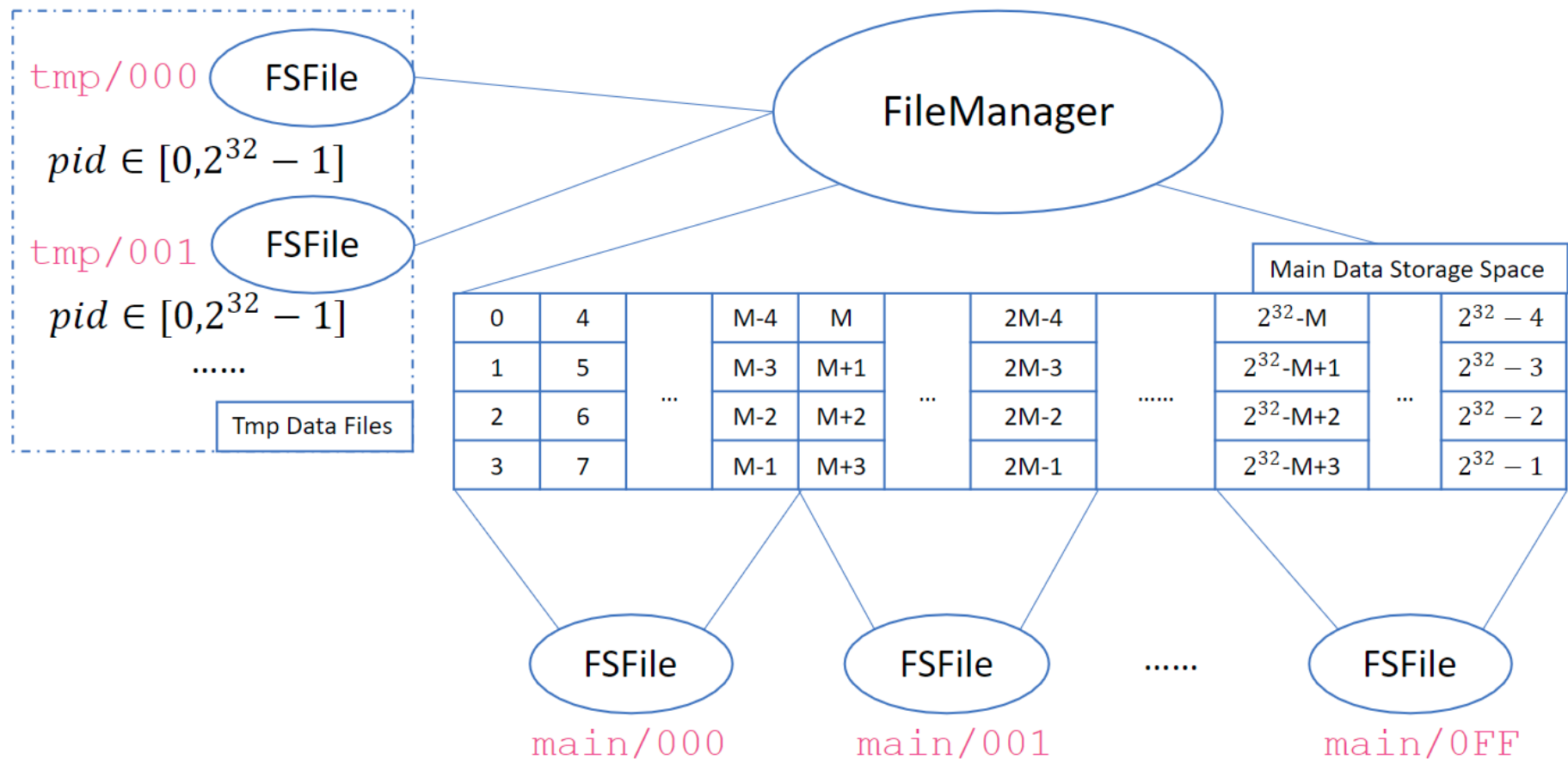


Figure 2: File Manager

Taco-DB manages its data in a flat main data storage space organized as fixed-size pages (see Figure 2), through the `FileManager` interface defined in `include/storage/FileManager.h`. The size of each page is fixed to `PAGE_SIZE` = 4096 bytes. The page numbers are represented as the unsigned 32-bit integer type `PageNumber`, where page number `0 (INVALID_PID)` is an invalid page number reserved for file manager use only. Recall that you have implemented an interface `FSFile` for file I/O over the file system. The file manager breaks up the entire space into up to 256 files stored on your file system (usually in `build/tmp/tmpd.XXXXXX/main` when you're running tests). There is no need to worry about the internals of the file manager, but it is useful to understand how the file manager manages the page for debugging purposes.

There is a single global instance of file manager `g_fileman` defined in `include/dbmain/Database.h`.

Note: the `FileManager` also manages the temporary data files, but each temporary file has its own page number space that is independent of the main data storage space. They are not used in this lab so we will cover the details in later labs that use temporary files. Within the context of this lab, a file page refers to a page in the main data storage space.

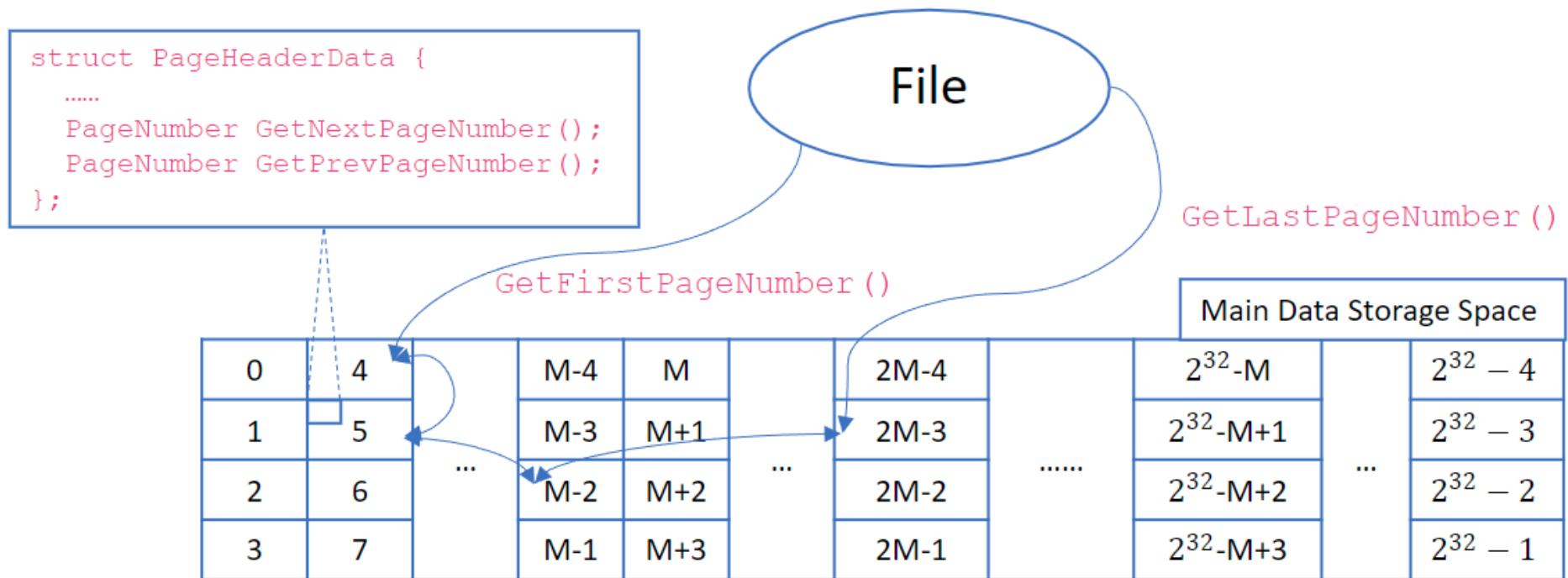


Figure 3: File manager managed (virtual) file

The file manager further divides the entire main data storage space as (virtual) files (see Figure 3). Each file is simply a linked list of pages in the main data storage space, which may be created and opened through the file manager and you may find the first page, the last page, or extend the file through the `File` interface. Each data page has a fixed-size header `PageHeaderData` at the beginning that is managed by the file manager, which stores the next and previous page number and some other information internal to the file manager. When you access a page in this lab, you must not modify the area occupied by the `PageHeaderData` outside the file manager.

Buffer Manager

Source files to modify:

- `include/storage/BufferManager.h`
- `src/storage/BufferManager.cpp`

While one could read/write pages directly through `FileManager::ReadPage()` and `FileManager::WritePage()`, it is not very efficient as each read/write incurs an I/O syscall into the operating system. As we covered in the lecture, DBMS utilizes a page buffer manager to keep a set of pages in the memory in order to reduce I/O syscalls and increase system throughput. In Taco-DB, the `BufferManager` is built on top of the `FileManager`. It handles all the page access requests from other system components and maintains a buffer of disk pages in a fixed number of buffer frames, by issuing I/O calls on the global file manager instance `g_fileman`.

There is a single global instance of buffer manager `g_bufman` defined in `include/dbmain/Database.h` except in buffer manager tests.

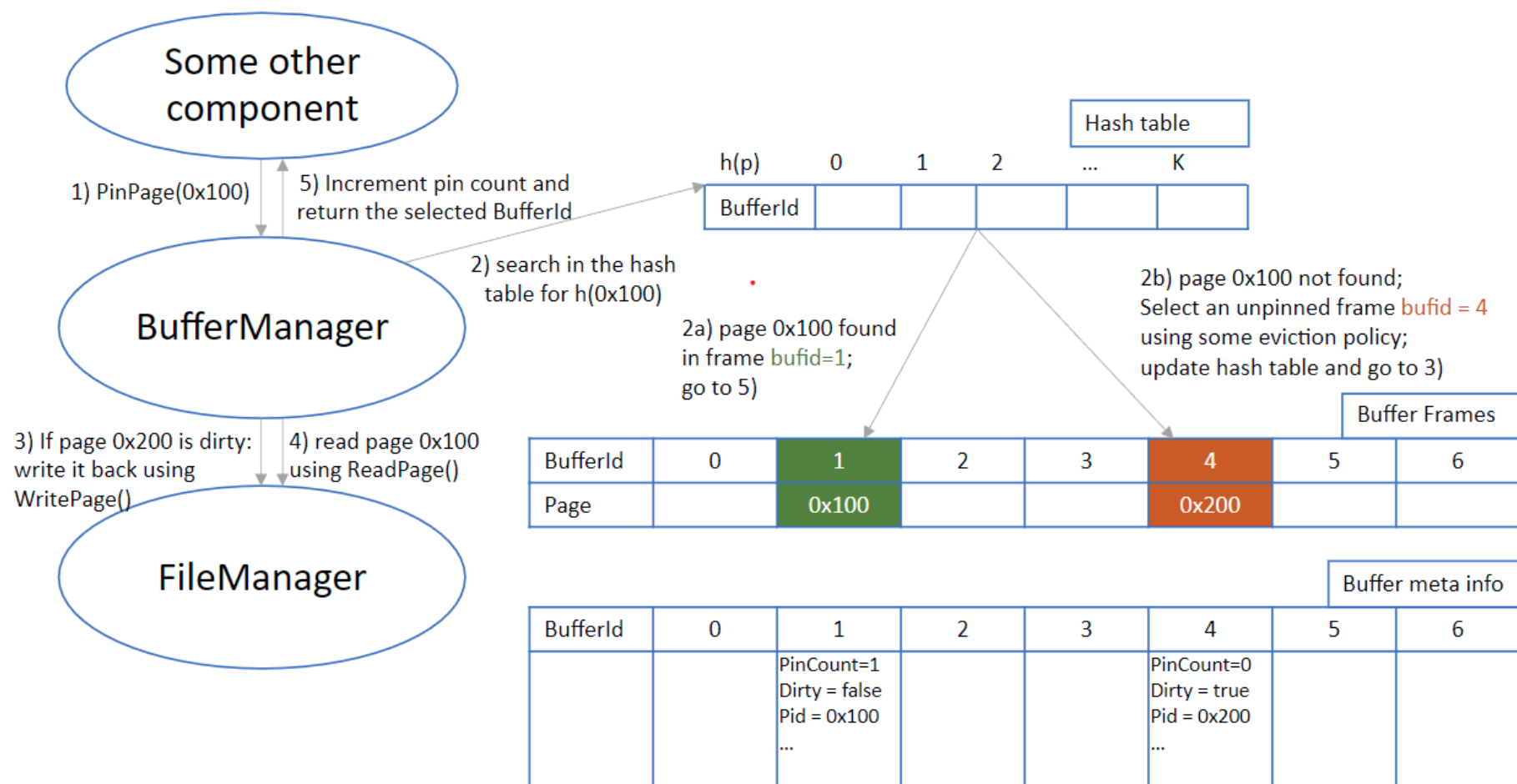
Figure 4: Buffer Manager `PinPage()` call

Figure 4 is a simplistic sketch of how BufferManager works. It allocates a number of consecutive pages in the memory indexed by an unsigned integer type `BufferId` from 0, with `INVALID_BUFID` defined as `std::numeric_limits::max()`. The buffer frames **MUST** be aligned to 512-byte boundaries (using `aligned_alloc()`), because the FileManager uses `O_DIRECT` I/Os over the file system. For each buffer frame, the buffer manager maintains some meta information in a separate structure, including the current pin count, a dirty bit, the page number its currently holding, as well as necessary information needed by the eviction policy. In addition, the buffer frame should maintain a hash table that maps page numbers to the buffer ids to speed up lookups.

Task 1: Add the necessary class members to BufferManager and implement:

- the constructor `BufferManager::BufferManager()`
- the destructor `BufferManager::~~BufferManager()`
- `BufferManager::Init()`
- `BufferManager::Destroy()`
- `BufferManager::MarkDirty()`
- `BufferManager::GetPageNumber()`
- `BufferManager::GetBuffer()`

Your code should still pass `BasicTestBufferManager.TestNormalInitAndDestroy` at this point (it passes without adding any code!).

Upon a `PinPage()` call, the buffer manager should look up the page number in the hash table. If it is found in some buffer frame, it increments the pin count and returns its buffer id. Otherwise, the buffer manager selects an unpinned buffer frame using some eviction policy (e.g., LRU, MRU, Clock, etc.) and writes back the page currently in the selected frame if it holds a dirty page. Then the requested page is read into the select buffer frame, which is returned to the caller with its pin count incremented. Note that a buffer frame may have a pin count greater than 1. An `UnPinPage()` call simply decrements the specified buffer frame's pin count.

Task 2: Implement `BufferManager::PinPage()` and `BufferManager::UnpinPage()` without a working eviction policy.

Your code is likely to pass the following test at this point:

- `BasicTestBufferManager.TestPinPageWithoutEviction`
- `BasicTestBufferManager.TestMultiplePins`
- `BasicTestBufferManager.TestUnpinPageNotPinned`
- `BasicTestBufferManager.TestPinInvalidPid`

Task 3: Implement a reasonably good eviction policy of your choice (e.g., LRU, MRU, Clock, etc.).

Implement a reasonably good eviction policy of your choice (e.g., LRU, MRU, Clock, etc.). Your code is likely to pass the following test at this point. Note that these tests do not test the efficiency of your eviction policy yet.

- `BasicTestBufferManager.TestPinPageWithEviction`

- `BasicTestBufferManager.TestNoAvailableBufferFrame`

The buffer manager in Taco-DB does not flush pages back unless the page is evicted or the database shuts down, as opposed to in a more sophisticated DBMS where the buffer pool may flush pages from time to time for performance and recovery consideration. In Taco-DB, there is a `BufferManager::Flush()` function that flushes all the dirty pages back to the disk. It should only be called during database shutdown from `BufferManager::Destroy()`. When that happens, all the buffer frames (whether it is clean or dirty) should have zero pin counts. It is considered as a bug if some pin count is not zero so it throws a fatal error in that case.

Task 4: Implement `BufferManager::Flush()` and add a call to it in `BufferManager::Destroy()`.

Your code should pass all the tests in `BasicTestBufferManager` and `BasicTestEvictionPolicy` at this point. The two tests in `BasicTestEvictionPolicy` checks if your eviction policy is working and reasonably good (i.e., not random eviction). They take a bit longer to run (tens of seconds in debug build), so their timeout is set to higher values on Autolab and in the offline tests.

Schema and Record Layout

The next thing you have to understand is how fields are laid out given a particular schema in Taco-DB and answer a set of questions in a writeup submitted along with your code. Specifically, you should read the following source files:

- `include/catalog/Schema.h`
- `src/catalog/Schema.cpp`

The first thing to understand is Taco-DB stores data and type information separately both in memory and on disk, and they are accessed differently. A `Datum` (see `include/base/datum.h`) stores an object of any type in memory as a sequence of bytes. You may think of it as a `union` of values of different types (e.g., integers, floats, strings), but internally it has different ways of storing things. In general, a fixed-length object of length 1, 2, 4, or 8 bytes is stored inline in a Datum object (e.g., `INT4`, `INT8`), while a variable-length object (e.g., `VARCHAR(5)`) or a fixed-length object of any other bytes in size is stored in a buffer and Datum keeps a pointer to it. Depending on where this buffer is located (on the buffer page or in some malloced memory), the buffer may be owned or not owned by the Datum. Note that we do not allow copying a Datum because of that. To reference a Datum, either use `const Datum&` or the `DatumRef` class. When an object is stored on a data page, it is always stored as a flat array of bytes without pointers or references. Note that a fixed-length object *may be stored as a variable-length datum* (e.g., `CHAR(5)`), which should be accessed using `Datum::GetVarlenBytes()`, `Datum::GetVarlenSize()` or `Datum::GetVarlenAsStringView()` rather than `Datum::GetFixedlenBytes()`.

On the other hand, Taco-DB relies on the type catalog information to interpret the values. The type information is looked up from the catalog table `Type`. These types are SQL types rather than C++ types, and they can be used to find information about how to manipulate these objects. There are a few important fields of the `Type` catalog table (`src/catalog/systables/Type.inc`) explained below:

- `typeid`: each type has a unique object id (Oid) in the system
- `typelen`: the length of a type if it is fixed-length and has no type parameter that may determine its length. If this is a fixed-length record and also has a type parameter that determines its length, this is undefined. If this is a variable-length type, this is always -1.
- `typisvarlen`: whether this type is variable-length
- `typalign`: a value of this type must be aligned to `typalign` byte memory address or within a data page. This is 1, 2, 4, or 8.
- `typlenfunc`: a function id such that the function accepts a type parameter (`INT8`) and returns its length if there's one. This is only valid when the type is fixed-length but its length is determined by its type parameter.

Finally, a built-in function is some C++ function that accepts a list of `Datum` of certain SQL types and returns a `Datum` of a certain SQL type. These built-in functions are managed by a centralized function catalog, and can be looked up/called dynamically through the `FunctionCall` method. For instance, the `typlenfunc` for a predefined SQL type is a built-in function that accepts a `UINT8` `Datum` and returns a `INT2` length. It calculates the binary length of this given type based on the type parameter. You may find the details of built-in functions in `include/base/fmgr.h` and `include/utils/builtin_funcs.h`. In later projects, we will add additional catalog information to identify functions that may operate on certain data types, such as numeric operations, string length and etc. For the task below, `typlenfunc` of `CHAR` returns the type parameter itself, i.e., the maximum length of the string.

Task 5: Read the implementation of Schema to answer the following questions in your writeup (please put it in a pdf as `lab1-writeup.pdf` under the root folder of your git repo).

Considering the following table schema:

```
CREATE TABLE A(  
  f1 INT2 NOT NULL,  
  f2 VARCHAR(5),  
  f3 CHAR(10),  
  f4 INT4 NOT NULL,  
  f5 VARCHAR(10),  
  f6 UINT8);
```

1. Draw a data layout diagram for record `{f1: -10, f2: 'abc', f3: 'abcde12345', f4: 15645134, f5: null, f6: 126}`. Please include

essential information including where each field is laid out (its byte range), important byte information stored in this layout, and meta-information stored along with the record (if any) on your diagram.

2. Briefly explain how the function `GetField(field_id, payload)` returns the data in a record corresponding field. Use extracting field 4 (`f4`), field 6 (`f6`) and field 2 (`f2`) from the record given in the first question as an example. Specifically, go through essential code logic on how offset and length of each field is acquired.

Data Page

Source files to modify:

- `include/storage/VarlenDataPage.h`
- `src/storage/VarlenDataPage.cpp`

How to run parameterized tests individually:

The tests for the data page are parameterized. You won't be able to run each individual test case separated if running them with `ctest`. To run a test with a specific parameter, e.g., the page initialization test with `min_reclen = 8, max_reclen = 8` and `usr_data_sz = 0`, run it directly using the test executable with the command: `./tests/storage/BasicTestVarlenDataPage --gtest_filter='*TestInitialize/8_8_0'`

Taco-DB manages its data in the fixed-size pages, where we may store a number of records as well as some additional information needed by the database. Taco-DB supports variable-length data types (e.g., `VARCHAR`), as well as nullable table fields, and thus it needs to have the capability of storing variable-length records on the data pages using a slotted data page design. In this lab, a `Record` is simply an array of bytes of a certain length. Note that a record **must** be aligned to some 8-byte address (you may use `MAXALIGN()` for that).

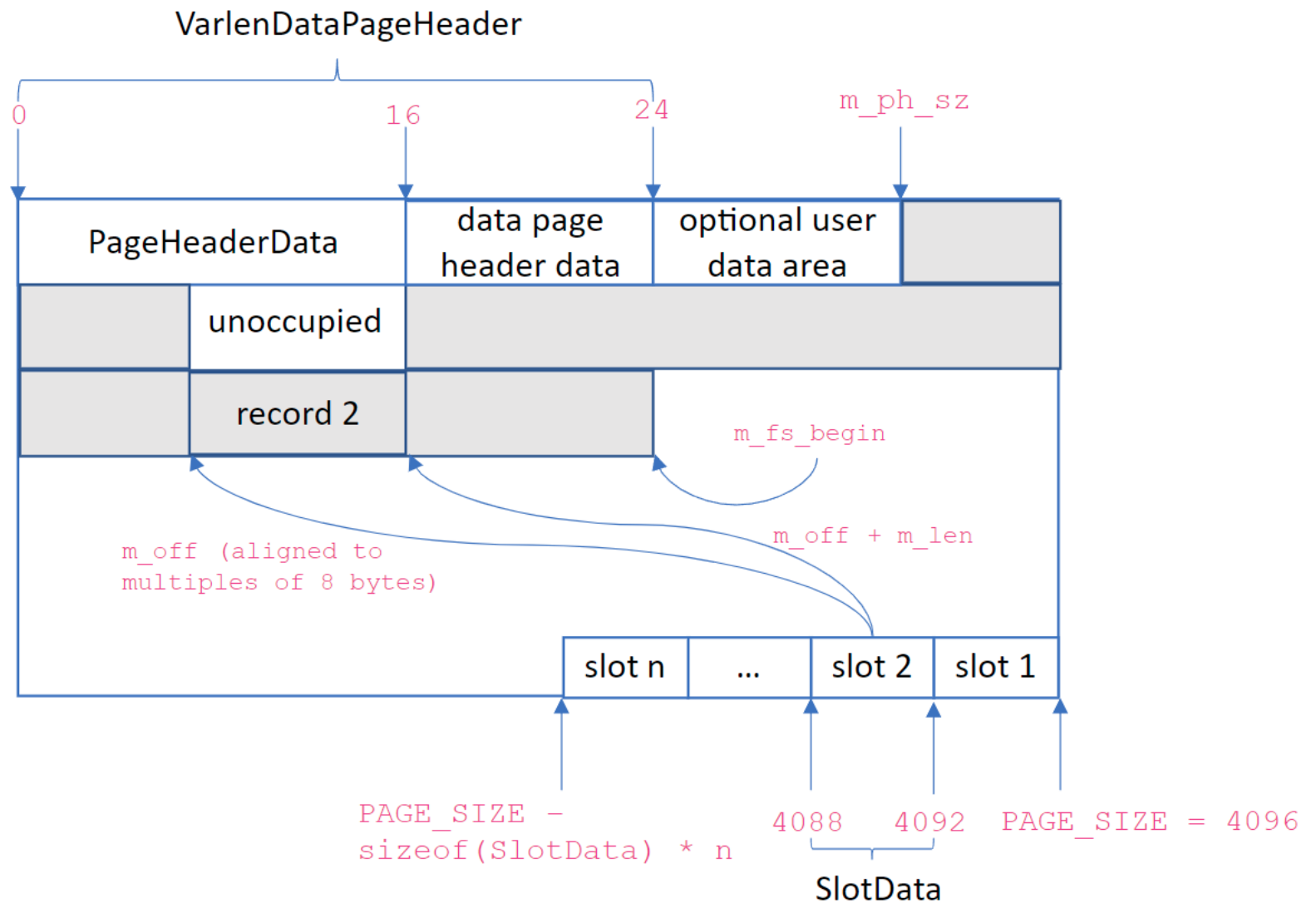


Figure 5: Data Page that supports variable-length records

The layout of a data page is shown in Figure 5. The page begins with a 24-byte header `VarlenDataPageHeader`, which includes the `PageHeaderData` at offset 0. Following that is an optional fixed-size user data area (you'll need it for the later B-tree file lab). Slots on the page are indexed as a `SlotId` type from 1 upwards (with `INVALID_SID=0`). The slot array starts at the end of the page and grows backward. Hence the `SlotData` of slot `sid` may be accessed as `((SlotData*)(pagebuf + PAGE_SIZE))[-sid]`, if `char *pagebuf` points to

the beginning of the page. The actual record payload for slot sid is stored somewhere between the end of the user data area and the beginning offset of the free space. To insert a record, find a free existing slot or create a new slot in the slot array, and find some empty space to copy the record payload. To erase a record, set its slot to be invalid and possibly remove all the invalid slots at the end of the slot array if any.

Task 6: Implement the page initialization and a few page meta information functions:

- `VarlenDataPage::InitializePage()`
- the constructor `VarlenDataPage::VarlenDataPage()`
- `VarlenDataPage::GetUserData()`
- `VarlenDataPage::GetMaxSlotId()`
- `VarlenDataPage::GetRecordCount()`

Your code is likely to pass the tests `*/BasicTestVarlenDataPage.TestInitialize/*` at this point.

Task 7: Implement the slot query and insertion functions: `VarlenDataPage::InsertRecord()`, `VarlenDataPage::IsOccupied()`, and `VarlenDataPage::GetRecordBuffer()`.

Your code is likely to pass the tests `*/BasicTestVarlenDataPage.TestInsertRecord/*` at this point.

Task 8: Implement the record erasure function: `VarlenDataPage::EraseRecord()`.

Your code is likely to pass the tests `*/BasicTestVarlenDataPage.TestEraseRecord/*` at this point.

Task 9: Implement the record update function: `VarlenDataPage::UpdateRecord()`.

Your code is likely to pass the tests `*/BasicTestVarlenDataPage.TestUpdateRecord/*` and `*/BasicTestVarlenDataPage.TestSidNotValidErrors/*` at this point.

As you might have noticed, there could be unoccupied spaces between offsets `m_ph_sz` and `m_fs_begin` (i.e., holes) after erasure or update. Hence, the data page should try to compact those spaces by moving the record payloads to be consecutive. This process is called page compaction. During page compaction, the `SlotData` in the slot array may have to be updated, but the slots themselves may not be moved. In other words, a valid slot remains valid and still refers to the same record; an invalid slot remains invalid, and the total number of slots remains unchanged after compaction. To avoid trying to compact a page every time an insertion or an out-of-place

update happens, you should maintain a bit `m_has_hole` in the page header to indicate whether it is possible to (not must) have holes in the occupied space.

Task 10: Implement the page compaction and update the page update functions accordingly.

You should pass all the tests in `BasicTestVarlenDataPage` at this point.

Hint: not implementing compaction correctly is highly likely to only fail the last three tests with compactions, so you might want to prioritize your effort appropriately to maximize your score.

Optional: there are two functions for maintaining a sorted array of records on the page: `VarlenDataPage::InsertRecordAt()` and `VarlenDataPage::RemoveSlot()`. These are not required in this lab and we do not provide the tests for them yet. You'll need to implement only in a later lab of implementing B-trees.

Heap File (Table Interface)

Source files to modify:

- `include/storage/Table.h`
- `src/storage/Table.cpp`

Tips on the tests:

There will be two versions of each test for the heap files: `BasicTestTable` and `SystemTestTable`. Basic tests are available in the handout, while the system tests are hidden. Their test code is actually the same (see `tests/storage/TestTable_common.inc`). The difference is that the basic test uses the `BootstrapCatCache` to create the table descriptor and thus do not have dependencies on your heap file implementation, while the system test uses the `PersistentCatCache` which initializes a catalog using your heap file implementation and thus is more likely to fail to initialize if your implementation is buggy. You might want to pay attention to the specifications in the header file in order to pass all the tests.

A heap file is simply an unordered (multi-)set of records. It is the most basic way of organizing records in DBMS. In Taco-DB, a heap file is implemented on top of the file manager managed virtual files and stores records in some arbitrary order in the list of pages. A table descriptor `TabDesc` stores essential information about a table, which includes the file ID of the table. The `Table` class should open the file through the global file manager `g_fileman`, make calls to the global buffer manager `g_bufman` to access the pages, and update or

retrieve records on the page using `VarlenDataPage`. It should support inserting, updating, deleting, and iterating over records. To simplify the implementation, you only need to allocate pages when there's no space for insertion and do not have to reuse free space on previous pages (but you could if you'd like to). In addition, you should make sure you deallocate a page when all of its records are erased. The `Table::Iterator` class provides a forward-only iterator over all the records, and it should only keep at most one page pin at a time.

Task 11: Implement the table initialization and construction functions:

- `Table::Initialize()`
- `Table::Create()`
- `Table::Table()`
- `Table::~~Table()`

Your code is likely to pass `BasicTestTable.TestEmptyTable` at this point.

Task 12: Implement `Table::InsertRecord()`, `Table::StartScan()`, `Table::StartScanBefore()` and the `Table::Iterator` class. You may change, delete, or add to the existing members in the `Table` and `Table::Iterator` if necessary, and the signature of any private functions – they only exist as a suggestion.

Your code is likely to pass `BasicTestTable.TestInsertRecord` and `BasicTestTable.TestStartScanBefore` at this point.

Task 13: Implement `Table::EraseRecord()`.

Your code is likely to pass `BasicTestTable.TestEraseRecord` at this point.

Task 14: Implement `Table::UpdateRecord()`.

Your code should pass all the tests in `BasicTestTable` and `SystemTestTable` at this point.

Submission Guideline

When you are ready to submit the lab, push your code to Github and find the latest commit hash. Git commit hash can be found on the Github website or through the command `git log`. Copy and paste the commit hash into the text box for this assignment.