Project Stage 1.0: Paginated Data Access

**Due date:** Tuesday, October 29, 11:59pm.

**Note:** Stage 1 of the project consists of two tiers: Stage 1.0 and Stage 1.5. **Both** are due October 29. The Stage 1.5 assignment will be released during the week of October 14 (either on Tuesday or on Thursday, depending on how far we proceed in the course).

## Overview

Stage 1 of the project incorporates the work you have done during **Lab 2** and extends it to build:

- Complete **buffer manager** layer.
- Functionality for **paginated disk access** for sequential and hashed DB files.
- Functionality for **indexing XML data** on disk.

This is done in two sub-stages, Stage 1.0 (this one) and Stage 1.5. Both have the same deadline, but Stage 1.5 assignment will be released after we cover the necessary material in class.

Additionally, there is a *synergistic activity* for Stage 1.0 and there may be one for Stage 1.5.

## ArferDB: Stage 1.0

This document describes the lower tiers of ArferDB architecture:

- Buffer Manager
- Page Read/Write layer
- DB file manager

While ArferDB is a *native XML DBMS*, its lower tiers implement the generic low-level DBMS functionality and prepare the "ground" for XML-specific layers that will use their APIs. As such, most of the implementation details for this part of the project still follow pretty closely the class discussions devoted to the architecture of the appropriate layers of the Relational DBMS.

## Overview of Data Management in ArferDB

Before providing specific instructions for the low-level ArferDB APIs, we discuss the general approach to data management in ArferDB.

ArferDB is a native XML DBMS designed to store collections of *static* XML documents and provide XPath-compatible query interface to them.

ArferDB supports the following repository and document-level operations:

- Create new XML repository/collection.
- Delete an existing XML repository/collection.
- Insert an XML document into an existing XML repository/collection.
- Query an existing XML collection with a query in XPLite, a simplified version of XPath, version 1.0 path expression language for XMl documents.
- Display a list of available XML repositories.

**XML repositories.** Each XPLite query is run against a single XML repository (collection) managed by ArferDB. An XML repository is a collection of XML documents, i.e., essentially, a forest of XML trees. XML documents belonging to the same repository need not share the same schema, origin etc — any group of XML documents can be treated as a repository.

**XML repositories and tinyFS.** tinyFS models a disk as a single file containing a preset number of disk blocks. Internally, tinyFS maintains a single directory with multiple files.

ArferDB stores information about a single XML repository in a collection of data files. Because tinyFS allows no directories, we assume that the name of a tinyFS file representing one of the data files for a specific XML repository follows roughly the following naming schema:

```
<Repository>-<FileName>.dat
```

where `<Repository>` is the name of the XML repository and `<FileName>` is the name of the data file in the repository. All files for all repositories will be stored in the "root" tinyFS directory.

**Native XML storage.** XML documents in a repository will be stored by ArferDB in two different ways. First, the actual XML content will be *shredded* into a persistent tinyFS file. This content will be used in creating the query output, and, occasionally, in confirming the results of the queries. Additionally, ArferDB will use *parent-child inverted index* structures to index the contents of the XML documents and make these contents amenable to search via XPLite queries.

The data files representing the primary XML store, as well as the *parent-child inverted index* utilize the following data file types:

- Sequential data file. Primary storage and part of the index structures are stored in sequential data files.
- Hashed data file. Internal organization of the XML indexes may (will) require hashing the data.

Our goal on Stage 1.0 is to develop all the functionality for support of these two data file types. Stage 1.5 is devoted to implementing the specific data files and index structures for storing native XML data in ArferDB.

## Disk Block Management in ArferDB

TinyFS maintains its internal `blockId` in the first bytes of each disk page. `tinyFS` block read and write routines require only this `blockId` to access the block.

ArferDB must maintain multiple data files and must be able to access specific pages within each data file in a direct way. For example, the first page of each data file (the *header page* should be easy to find and refer to in ArferDB code. Because of this, we adopt a separate from tinyFS way of addressing tinyFS disk blocks.

Note that ArferDB needs to access only the disk blocks that belong to the data files it maintains. Because of this, we adopt the following scheme for addressing data file pages in ArferDB. Each ArferDB data file page will have an internal ArferDB-maintained address that consists of two parts:

- The unique identifier of the data file: e.g., a file name or a tinyFS `fileID`;
- An ordinal representing the unique id of the page inside a file.

To make life easier, define a C `struct` representing an ArferDB page address. A sample definition can be

```
typedef struct  {
    int fileId; /* tinyFS blockId of the file's iNode */
    int filePage; /* Page Id within the file: 1,2,3,.... */
} ArferDBAddress;
```

In what follows, `ArferDBAddress` refers to the `struct` specifying a blockId in the ArferDB frame of reference. The specific definition/implementation of this type is left to each team.

Note, that the proper maintenance of ArferDBAddresses is one of the tasks you will have to implement throughout the lower layers of ArferDB.

### Buffer Manager

The buffer manager layer for ArferDB is an extension of your **Lab 2** buffer manager. Essentially, in addition to the already implemented functions that use tinyFS block ids to exchange blocks between the tinyFS disk and the buffer, you will need to implement the versions of the functions that use the `ArferDBAddress`-style block ids. This is necessary to protect upper layers from the internals of the "operating system" (which, in our case, is represented by tinyFS).

Recall the API you needed to build for **Lab 2**.

| function declaration | brief description |
|---|---|
| `int commence(char * Database, Buffer * buf, int nBlocks);` | initialize the buffer |
| `int squash(Buffer * buf);` | end the work of the buffer manager |
| `int readPage(Buffer * buf, int diskPage);` | read access to the disk page |
| `int writePage(Buffer *buf, int diskPage);` | write access to the disk page |
| `int flushPage(Buffer *buf, int diskPage);` | flush the page from buffer to disk |
| `int pinPage(Buffer *buf, int diskPage);` | pin the page |
| `int unPinPage(Buffer *buf, int diskPage);` | unpin the page |
| `int newPage(Buffer *buf, int * diskPage);` | add a new disk page |

The additional functions you need to implement are:

`int readBlock(Buffer *buf, ArferDBAddress page):` this function works the same way as `readPage()` does, except the page address is the `ArferDBAddress` value.

`int writeBlock(Buffer *buf, ArferDBAddress page):` works the same way as `writePage()`, except the page address is the `ArferDBAddress` value.

`int flushBlock(Buffer *buf, ArferDBAddress page):` works the same way as `flushPage()`, except the page address is the `ArferDBAddress` value.

`int pinBlock(Buffer *buf, ArferDBAddress page):` works the same way as `pinPage()`, except the page address is the `ArferDBAddress` value.

`int unPinBlock(Buffer *buf, ArferDBAddress page):` works the same way as `unPinPage()`, except the page address is the `ArferDBAddress` value.

`int newBlock(Buffer *buf, char * fileName, ArferDBAddress * page):` this function creates a new disk block belonging to a given file (`filename`). It places the ArferDB address of the block in the `page` attribute[1] before returning.

[1] You can implement this function with `page` being passed as `ArferDBAddress * page` or as `ArferDBAddress ** page`. Here, and in all similar situations, I will use a single pointer for simplicity. Single-pointer functions require

`int setUpDisk(char * Database, int nPages):` This function is essentially a wrapper around the underlying tinyFS function `OpenDisk()`. It is included here to make sure that ArferDB has its own function call that sets up an empty disk. This function takes as input a name of the ArferDB database, i.e., the name of the tinyFS file (`char * Database`), and a number of disk pages (`nPages`). It works as follows. If a tinyFS file with the given name already exists, the function does nothing, and returns an appropriate exit code indicating the `"Database exists"` message. If the file with the given name does not exist, the function creates a new tinyFS emulated disk with the given (`nPages`) number of pages.

`setUpDisk()` also sets the created tinyFS file for use as an ArferDB disk, This may mean a lot of different things — including creation of special types of disk pages and filling those pages with preliminary information, setting up some internal data structures in the created disk, and so on. I expcet `setUpDisk()` to be a function that evolves with your implementation of `ArferDB` over time, as you discover what needs to be put on the tinyFS disk upon its setup.

Notice that `setUpDisk()` belongs to the buffer manager layer only because there is no lower layer that you are implementing in the project. The function itself shall not use the buffer (note that no buffer parameter is passed to it): in fact, this function will usually be called when you need to prepare for launch of the buffer manager with a new data store. The intended use is something like this:

```
char * dbName = "myDatabase";
int nPages = 5000;
Buffer buf;
bufferSize = 200;

....

int flag = setUpDisk(dbName, nPages);
commence(dbName, &buf, bufferSize);
```

Under normal circumstances, `setUpDisk()` shall always be run before `commence()` runs. Because of this, you can alter the `commence()` function to only initialize buffers when the given tinyFS disk already exists (this ensures that you can control the size of the tinyFS disk from ArferDB and/or any debugging/testing program).

### Buffer Data Exchange Layer

The Buffer Manager layer manages disk pages as whole indivisible objects. It does not actually alter the contents of the pages, nor does it provide any functionality for the upper layers of the

you to `malloc()` or `calloc()` the appropriate pointer in the calling function, before making the designated function call. Double-pointer functions can allocate the space for the parameter in their own body.

code to access the specific data stored on the page. This responsibility lies with the next layer of functionality, the Buffer Data Exchange layer.

The Buffer Data Exchange layer has only two functions that you need to implement:

**`int read(Buffer *buf, ArferDBAddress page, int offset, int nBytes, unsigned char * data):`** this function takes as input the buffer (`buf`), the ArferDB address `page`, the location on the page, `offset`, and the total number of bytes to be read, `nBytes`. It works as follows.

- First, the function will read the requested page into the buffer/find where the page is in the buffer.
- Second, the function will access the contents of the page, and will read `nBytes` bytes of data starting at the position `offset` on the page. It shall place the collected data into the `data` container — the last parameter of the function.
- The function shall return a status code: either success, if the read successfully filled the `data` variable with the right amount of data, or one of the possible errors:
  - Unable to place the page in the buffer.
  - Disk page does not exist.
  - Incorrect starting position (offset).
  - Page overrun (i.e., reading `nBytes` bytes of data starting a position `offset` will result in going past the boundary of the disk page.
  - Buffer not initialized.
  - Not enough space in the receptor: somehow `data` has been initialized improperly and does not have `nBytes` of size to accommodate the input read.

**`int write(Buffer *buf, ArferDBAddress page, int offset, int nBytes, unsigned char * data):`** this function takes as input the following parameters:

| | |
|---|---|
| `buf`: | (pointer to) the buffer data structure; |
| `page`: | address of the disk page to which the write has to be made; |
| `offset`: | the location at which the write commences; |
| `nBytes`: | number of bytes to write; |
| `data`: | data to be written to the page. |

The function proceeds as follows:

- First, it ensures that the disk block referred to by the `page` parameter is in the buffer, and, if not, brings it in.
- Second, it accesses the contents of the page. It finds the position `offset` on the page, and writes `nBytes` bytes from the `data` array to the block in the buffer[2]

[2] Note that the `data` array can technically contain more data in it, then what needs to be written.

- Third, it "closes the books" by performing the buffer write operation.

**Note:** Any actual write to a page in the buffer must ensure that the data "falls" where needed. Therefore, before actually placing new bytes on the buffer page, you **must always pin** the page. Similarly, once the write is over, you can unpin the page.

The function returns a status code: either success, if the write operation went through as expected, or one of the following error codes:

- Unable to place the page in the buffer.
- Disk page does not exist.
- Incorrect starting position (offset).
- Page overrun (i.e., writing `nBytes` bytes of data starting a position `offset` will result in going past the boundary of the disk page.
- Buffer not initialized.
- Not enough data to write: somehow `data` has been initialized improperly and does not have `nBytes` of data in it to write to the buffer page.

## Data File Layer

The ArferDB layer responsible for data file maintenance is probably one of the largest in the system. As specified above ArferDB needs to store multiple data files per single XML repository it manages. All[3] data stored in ArferDB can be accommodated in two types of data files:

- sequential files: most of the primary structures of XML storage will store XML documents in the depth-first search order of the element appearance in the document, and will pack XML documents into a repository one after another. Some portions of the secondary index structures may also wind up being stored sequentially.
- hashed files: some parts of the secondary structures can be conveniently stored in a hashed data file.

As of the moment, the use of heap files in `ArferDB` is not planned.

In addition to implementing the operations for creating and maintaining sequential and hashed data files, you need to make sure that both types of file can work with both *fixed length records* and *variable length records*. In particular:

[3] Because the full specification for ArferDB does not exist yet, "all" should be interpreted as "all I know of at the moment".

- *fixed length records*: the primary structure of storing (shredding) XML to disk can be built using fixed length records if we adopt some reasonable restrictions on the input XML files[4]
- *variable-length records with repeating fields:* these will be the bread and butter of our secondary index structures.
- *variable-length records with variable length fields:* some parts of XML documents, namely text content (PCDATA) of XML elements and the values of XML attributes (CDATA) can be of variable and unlimited length.

This potentially leads to six different data file architectures:

1. Sequential file with fixed length records.
2. Sequential file with variable-length records with repeated fields.
3. Sequential file with variable-lenght records with variable-length fields.
4. Hashed file with fixed length records.
5. Hashed file with variable-length records with repeated fields.
6. Hashed file with variable-lenght records with variable-length fields.

A single ArferDB implementation might not need to instantiate files of all six types (because selecting the type of some of the data files is an either-or proposition), however, each team may need at least 4 different types of files from the list above: both types of files for fixed-length records and at least one type of file for different variable-length records.

### Common Specs

All data file implementations must accommodate the following requirements.

**Header Pages.** Called `INODE`s by tinyFS, these are the first pages of each tinyFS file. You need to determine what information needs to be stored in the header page, and build the support for maintaining the content on the header page of your data file current throughout your implementation.

**Data Page organization.** Each data file must have at least one type of *data pages*. Some data files may allow for more than one type - your implementation must be flexible. Specific design decisions will need to be made by each team once we discuss all ArferDB data structures. However, the data file layer must be flexible to accommodate multiple types of data pages if needed.

**Page Headers.** Each page must have a page header. Page header sizes and structures may differ between different types of data files: you will be putting different data into the page header for a hashed file than you will for a sequential file. You need to try to keep the size of the page header for

[4]These will be provided to you in the Stage 1.5 description.

all types of disk pages in the same file (except for the INODE page) to be the same. Page headers must also account for the space reserved by tinyFS for its own needs and **NOT overwrite it**.

*Advice:* make sure you have enough space in your page header for growth. DBMS design often leads to the needs to put more information somewhere in the data files, and page headers are often the appropriate places to store it. Leave yourselves space to grow.

**Record lenghts.** Your implementation shall be able to accommodate records of any length (both in the fixed length mode, and in the variable length mode). Except for variable-length attribtes in some records, which can be really large in size, all other records will have sizes that are smaller than the disk page space reserved for data storage. Most records will have at least one or two `int` values stored in them, so you can assume that 8 bytes is the shortest possible record length.

### Common Functionality

The following operations must be implemented for every type of data file. Unlike the lower levels of ArferDB, where the function specs were provided for you, and unlike some of the upper layers, where we will do the same, the specific function specs for this layer are left for each team to design and implement. Please use sound design principles when designing your function libraries/APIs.

**Creation of a new data file.** The function creating a new data file, shall take as input the file name, and three attributes controlling the file generation process:

| | |
|---|---|
| file type | *hashed* or *sequential*; |
| record structure | *fixed length*, *repeated attributes*; |
| number of types of data pages | |

Based on this information, the function shall initialize the new file in tinyFS, create all necessary disk pages for the empty file to be ready for future use. Each team controls essential decision-making on what specifically needs to be done to achieve this.

You may want to provide some more information about the intended use of the data file. For example, each data page type can be associated with a type of a record to be stored there:

- fixed-length record
- record variable length attribute
- record with repeated attributes

In addition, for fixed length records, the record length can be provided ahead of time, while for variable-length record, the *minimum record length*, i.e., the length of a the fixed part of the record, can be provided as well. So, your function can take as input an array of page type descriptors that convey this information.

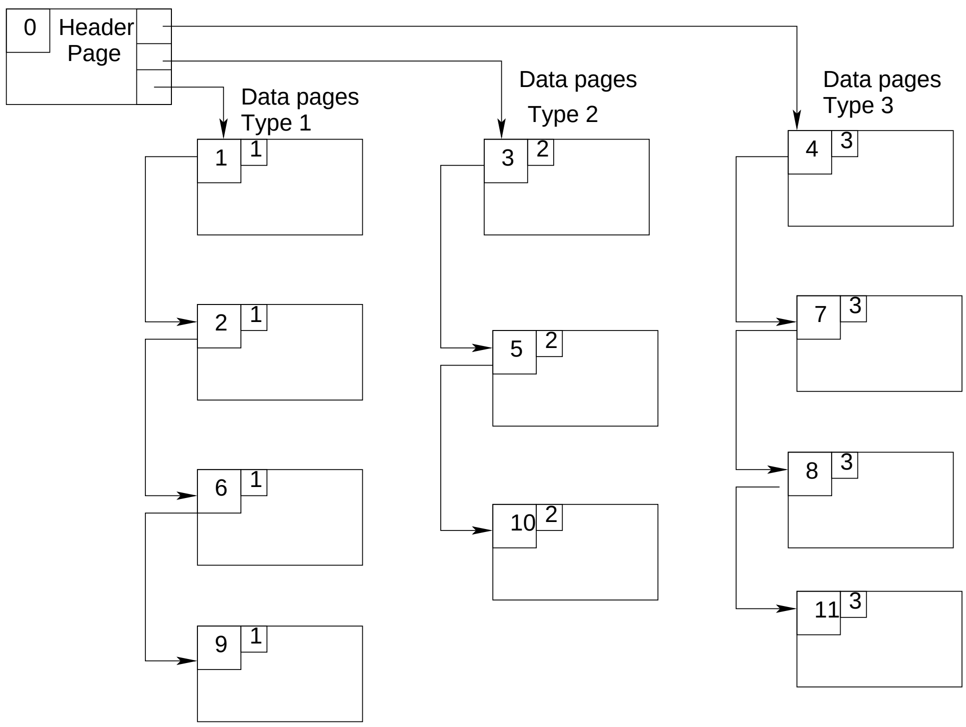


Figure 1: Design of a sequential file for multiple types of data pages.

**Deletion of a data file.** Generally speaking, tinyFS handles file deletion, and deletion of ArferDB files can only happen when an XML repository is being deleted, so this function should really be just a pass-through to the underlying tinyFS file deletion function. It should be implemented inside ArferDB for the sake of symmetry.

**Creation of a new data page.** This function shall take as input the *data page type* to specify what type of a data page to create. There actual implementation of this function is different for sequential and hashed files. See the specifics below. The function should make the `ArferDBAddress`-style page address available to the calling function for future use.

**Deletion of a data page.** This operation will be used rarely, as ArferDB does not actually delete data from repositories. But it should be implemented for the sake of completeness and because it will make debugging easier.

**Sequential Files**

The overall structure of an ArferDB sequential file is shown in Figure 1. Essentially, sequential files may contain data pages of different types. Each type of a data page represents different record structures that need to be stored. The sequential fille essentially maintains all records of the same type (i.e., all disk pages of the same type) in a single list, and it maintains all the lists in parallel[5].

**Record headers.** You need to determine ahead of time the structure of a record header for *any* record that is being iserted into a sequential file. Note that the structure of the record header can be different for the three different types of records, but all fixed-length records must have the same record header (if any), all variable-attribute length records must have the same record header (if any) and so on. For the purposes of this stage, what you need to know is **the length** of the record header for each type of record. This is important for all operations.

The following functionality, specific to managing sequential files, needs to be implemented.

**Record insertion.** To make it easy to insert records into sequential data files, we need to expose the keys used to store them the the data file storage layer. We make the following assumptions:

- A single type of data page will always have keys of the same length[6].
- The record insertion function will take as input the following information:
  - type of data page into which to insert the data.
  - value of the key.
  - record to insert.
  - If the record is of variable length, then also, length of the record.

  We assume, that the input record *has the structure as depicted in Figure 3*: starting with a fixed-length (known to you) record header, followed by the key value, followed by the rest of the record.

**Record updates.** We assume that the record update function provides the ArferDB address of the record. For fixed-length records, updates are straightforward - replace the records on the page. For variable-length records, ensure that you perform a correct *record shift* to accommodate potentially different size of the new record (this may involve creating a new page)[7]. For variable-length records, the length of the new record shall be provided as an input parameter to the function.

[5]The figure shows a single linked list, but in reality, you want a double linked list there to make sure you can navigate the pages in both directions, and to be able to insert new pages in the middle of the list.

[6]This might come to bite us later when we create parent/child indexes that index text, but we will deal with this issue when we get there.

[7]While primary structures for XML data storage will not be subjects to updates, the index structures will be, so update functionality is important.

| | | |
|---|---|---|
| Record Header | Record Key | Remainde of the record |

Figure 2: Design of a sequential data record.

**Record deletion.** The record deletion function should receive an ArferDB address of the record as input. It shall delete the record and "collapse" the disk page after the deletion is made (the data page invariant after insertion and deletion operations is that free space is at the end of the page). For variable-length records, the length of the new record shall be provided as an input parameter to the function.

**Key search.** Implement a naïve key search in a sequential file. Given a key value and a page type, search the data pages of the specified type for the record(s) with the specified key value (recall - key values are not necessarily unique). A full scan is the simplest method here, but you can also try binary search, although for this to work you may need to instrument a few more data structures inside your data files. The key search function should populate a list of ArferDB addresses and/or retrieve the actual content of the records (you can implement two key search functions: one for each type of output - you will find them useful later).

**Handling fixed-length vs. variable-length records.** It is worth mentioning this one more time. If you are handling variable-length records (of any type) in your code, make sure that you pass the length of the record as an input parameter to any function you call. You can either pass this information as an extra input parameter, or you can encode it in, for example, the first two bytes of the `unsigned char *data` bytearray.

## Hashed Files

The overall structure of a hashed file is shown in Figure **??**. A hashed file stores a single hash table of records. Under most normal circumstances, only one type of data pages is used in hashed files[8].

**Record headers.** Similarly to sequential files, you need to know the length of the record header for the hash table records of each type.

**Record insertion.** To insert a record into a hashed file, the record's key must be exposed. We assume that all ArferDB records are hashed based on a single-attribute key. We also assume that

[8]We may find exceptions later down the road, where we need some form of a "quasi heap file" data pages to archive some information. But even in this case, the actual hash table will be built out of pages of only one data type, and all other types of data pages will simply be pointed by the records from the hash table. For the current implementation, we allow to initialize hashed files with multiple data page types, but your code will only work with pages of one type - hashed file pages. All other page management would have to be done in the upper layers.

the key has a fixed length[9]. The structure of a hashed data file record is the same as the structure of a sequential file record (see Figure 3). The insertion function takes as input the key value, the full record to be inserted, the length of the record (for variable-length records of any type). The function should hash the key value, and proceed with the standard hashed file insertion procedure from there.

**Hash function.** Come up with your own. Be smart about it.

**Number of hash values.** Use it as a parameter when initializing the hashed data file.

**Record updates.** This will be a common operation, as hashed files are used primarily in indexes, indexes are built incrementally, and some incremental building of indexes will involve record updates. Record update function shall take the ArferDB address of the record and the new contents of the record as input (also, the length of the new record, if variable-length records are stored). For fixed-length records, update is a simple overwrite of the contents of the record. For variable-length record, make sure to shift the contents of the page to correctly accommodate the new record size.

**Record deletions.** Needed for the sake of completeness. The function should take an ArferDB address of the record as input. It shall delete the record and "collapse" the disk page after the deletion is made (the data page invariant after insertion and deletion operations is that free space is at the end of the page). For variable-length records, the length of the new record shall be provided as an input parameter to the function.

**Key search.** Implement a simple key search algorithm. Given a value of the key, hash it, and scan all disk pages for the computed hash value for records with the given key. Recall that (a) there may be more than one record with the same key value, and (b), there is no order in the hashed file pages, so in order to find **all** records with a given key value, you must scan all pages for the given hash key. The key search function should populate a list of ArferDB addresses and/or retrieve the actual content of the records (you can implement two key search functions: one for each type of output - you will find them useful later).

**Note:** Key searches are to be implemented for your convenience - this is a great debugging tool to confirm that you got the insertions/updates/deletions right.

## Deliverables

The full set of deliverables for Stage 1 of the project will be announced with the Stage 1.5 handout.

However, we can specify some deliverables even now.

[9]Which also may come to bite us later.

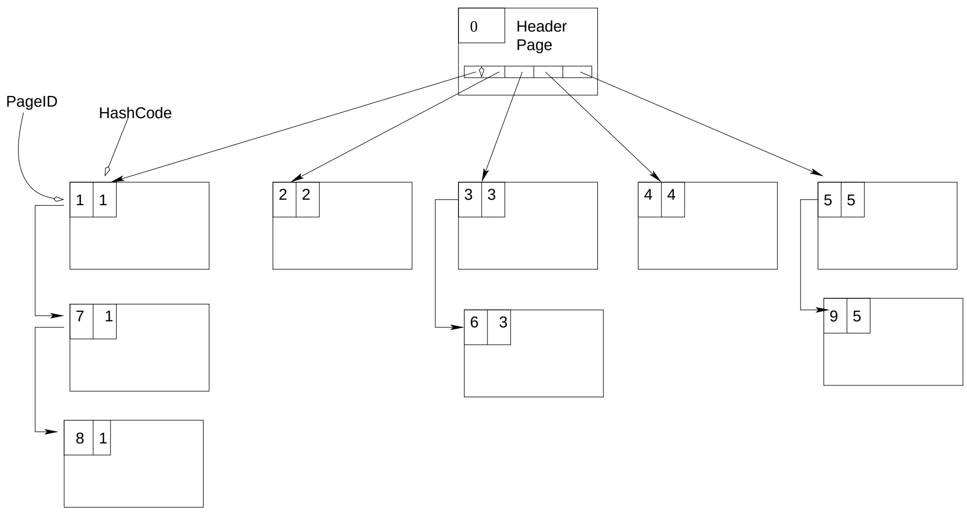


Figure 3: Structure of a hashed data file.

**Demo.** At the end of each stage, each team will perform an in-person demo to the instructor. The demos may occur during a lab period, or outside of classes, at an agreed upon time. (Presence of a full team is not required - we might be doing a demo in an otherwise crowded environment, but I'd like to see at least two people capable of "driving" the demo). Plan for 15-20 mins for each demo. The specifics of what to demo will be released with the Stage 1.5 specs.

**Design documentation.** Starting this stage, each team will maintain an ArferDB design document. The document shall keep track of all design decisions the team made. In particular, on this stage of the project, each team needs to document, at least the following decisions:

- Exact specifications, complete with explanations, for each major data structure/data type used in the development of ArferDB. (this is for the data types that map to the specific parts of ArferDB itself...)
- Exact specifications for all C functions developed that either (a) overrode instructor's specs, or (b) were under-defined in the instructor's specs.
- Specifications for all file header page content for all data file types.
- Specifications for the structure of all data page headers.
- Any other design decisions your team made.

At the end of each stage you will submit a current draft of the design document. The document shall be formatted as a technical white paper (you can use googledocs or something similar to maintain the working version of it), shall contain a title page with the class name, project name, team name, and the least of team members and their email addresses. It shall be written as a coherent narrative - *please no unannotated bullet-point lists or code fragments.* Soft copies of the submitted drafts shall be placed on the team's wiki page on The-Mothership repository. I will also ask for hard copies to be submitted to me during the class on the due date.

**Code snapshot.** By due date, you will submit a current snapshot of your code, complete with a `README` file explaining compilation and running details. The snapshots shall be submitted using `handin` (specifics in Stage 1.5 spec). Because the demos can span over a course of a week or more than a week, each team will do the demo "out of" the snapshot submission. Make sure you have test programs in your submissions that can be demoed (both as source code, and as an executable) in your snapshot submission.

**Github repository.** While there is no requirement that your team's github repository is used for development, you will also submit the snapshot of your code to the github repository. This is for archival purposes (and in case the `handin` demo does not work).

# Good Luck!