RFC: Page Buffer VFD Overview

Kyle Lofthus kyle.lofthus@lifeboat.llc
Aijun Hall aijun.ahhl@lifeboat.llc
John Mainzer john.mainzer@lifeboat.llc

This document described Page Buffer VFD implemented for enabling data encryption in HDF5.

1 Introduction

The purpose of the Page Buffer VFD is to convert random I/O requests to paged I/O requests. This ensures only paged I/O for the Virtual File Drivers (VFDs) lower in the VFD stack, which guarantees same size data I/O requests in the form of pages.

The initial implementation of the page buffer will use a hash table[1] to index pages currently in the page buffer. Multiple replacement policies will be supported (FIFO and LRU currently)[2]. The next sections provide implementation details.

2 Operations

1. Random I/O requests can be thought of as having three parts:
    1) **Offset**: Distance from beginning of file, in bytes.
    2) **Length**: Size of the request, in bytes.
    3) **Buffer**: For the incoming or outgoing data.
2. To convert random I/O to paged I/O, a random I/O request will be broken into at most three pieces, though not all three will appear in every I/O request.
    1) **Head**: Starts somewhere within a page and ends at or before the next page boundary. This is when the I/O request's offset starts in the middle of a page.
    2) **Middle**: Starts and ends at a page boundary. This will often be the largest section of the I/O request and contains all the full pages that make up the I/O request.
    3) **Tail**: Starts at the page boundary and ends somewhere in the middle of a page. This is typically for the end of the I/O request when it spills over a page boundary but does not fill the entire page.
3. Handling Heads and Tails:

[1] An image and description of a hash table is shown on the following page.

[2] Descriptions are on page 3 in Replacement Policy section.

1) The entire page that contains the Head or Tail must be loaded into the page buffer from file, if it is not already present, and the I/O request will be applied to the target page.
2) Write requests additionally mark the page as dirty[3].

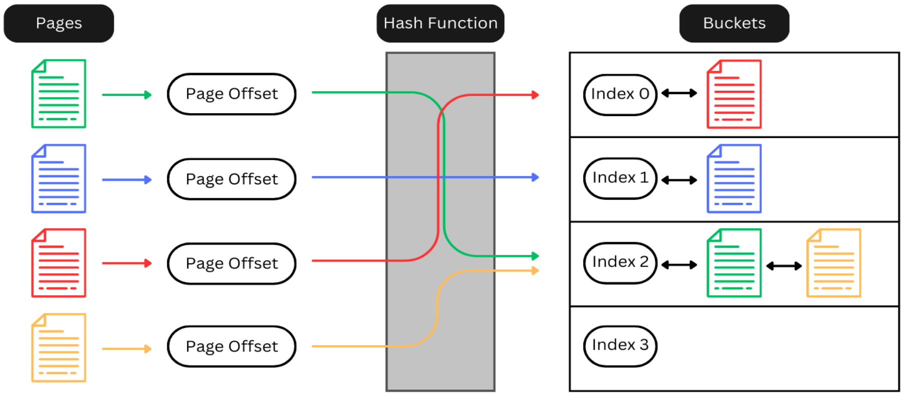
4. Handling Middles:
    1) **Read requests**: Any pages in the request that reside in the page buffer are satisfied from there, and the remainder are read directly from file.
    2) **Write requests**: Any pages in the request that reside in the page buffer are invalidated[4], and all pages in the middle are written directly to file.

Due to the number of pages in the page buffer being limited, loading a page may require a flush[5] and eviction of an existing page. Selection of the pages to be evicted is handled by the replacement policy.

3 Hash Table

A hash table is a data structure that allows for fast retrieval. Items stored in a hash table have a 'key-value' pair. Using a hash function, the item's 'key' is converted into an index in an array, where the value is then stored. A collision is when multiple items' 'key's' are converted to the same index. Chaining is a common technique used to handle collisions, by storing the colliding items in a Doubly Linked-List[6] at that index.

Figure 1: Hash table is used for fast retrieval pages of the pages



We call the keys for our Hash Table 'Hash Codes' and the values are pages being stored in the page buffer.

[3] A page marked 'dirty' denotes that page as containing modified data and needs to be written from the page buffer to the file.

[4] A page marked 'invalid' denotes that page as containing older data from what the file currently has and will not be written to the file.

[5] A 'flush' or flushing a page is when a dirty page or pages are written from the page buffer to the file.

[6] DLL: a linear data structure made of nodes to store data with next and previous pointers to travel forwards and backwards in the list.

Hash Function:

- Page size is fixed and must be a power of 2
- Hash Table's size is a power of 2
- Page Address is taken and the bottom bits[7] are cut off.
- Right shift the remaining bytes, this gives us the Page Number.
    - This is more efficient than using division to get the Page Number.
- Finally, the Page Number is taken modulo by the number of Hash Table Buckets to determine which bucket the Page is stored in.

3.1 Replacement policy

The current iteration of the Page Buffer is designed to support First-in-First-Out (FIFO) and Least-Recently-Used (LRU) replacement policies, with LRU being the current default. A DLL is used as the structure that controls the replacement policy.

1. **FIFO**:
    a. Page is added to the Page Buffer
        i. the Page gets added to the Hash Table as described above and gets added to the Head of the Replacement Policy DLL.
    b. When a Page is evicted, it is removed from the Tail of the DLL.
        i. This guarantees the pages are evicted in the order of the First-in-First-Out.

2. **LRU**:
    a. Same as FIFO.
    b. Any time a Page is accessed inside the Page Buffer the Page is removed from the DLL and inserted at the Head of the DLL.
        i. This process keeps Pages that are accessed more frequently in the Page Buffer longer for quicker access.
    c. When a Page is evicted, it is removed from the Tail of the DLL.

3. **Exceptions**:
    a. When a Page is invalidated, it gets added to the Tail of the DLL.
        i. This is done so an invalid Page will be the next Page evicted to remove outdated data as soon as possible.

[7] Example: assume our page size is 4096 bytes (2^12 bytes), the rightmost 12 bits in the address will be cut off.

Acknowledgement

This work is supported by the U.S. Department of Energy, Office of Science under award number DE-SC0024823 for Phase I SBIR project “Protecting the confidentiality and integrity of data stored in HDF5"

Revision History

August 2, 2024:	Version 1 circulated for comment within Lifeboat, LLC.
August 27, 2024:	Version 2 incorporates feedback and checked into GitHub repo.
September 20, 2024:	Version 3 adjusts wording and corrects pointers in FIFO and LRU description.
October 17, 2024	V*ersion 4* adds copyright symbol to footnotes.