# RFC: VFD SWMR

**Vailin Choi**
**John Mainzer**
**David Young**

The purpose of the SWMR (Single Writer Multiple Reader) feature is to allow a second process to read a HDF5 file while data is being written to it. Use cases range from monitoring data collection and/or steering experiments in progress to financial applications.

The existing SWMR implementation touches most parts of the HDF5 library, and therefore presents significant maintenance issues. Further, it offers no guarantees on the maximum time from write to read – which makes it problematic for some applications.

The primary impetus behind VFD SWMR is to implement SWMR in a more modular fashion – thus minimizing maintenance costs. Fringe benefits include allowing the HDF5 library to make guarantees for the maximum time from write to availability of data for read (subject to the performance limits of the underlying file system, and presuming that the writer calls the HDF5 library frequently), and the possibility of extending SWMR to NFS and object stores.

## 1 Introduction

The existing implementation of SWMR uses the strict write ordering and the atomic write guarantees of POSIX I/O semantics to ensure that the reader always sees consistent metadata. For example, if a B-Tree must be modified, all modified nodes below the top-level point of change are first duplicated, modified as required, and written to disk. This done, the top-level (or root) node of the change is over written in a single atomic operation. The original versions of the modified nodes below the root are retained for a time so that a reader traversing the B-Tree will see a consistent (but out of date) version of the B-Tree.

This approach requires that the metadata cache clients perform the necessary reallocations of metadata and specify the necessary write ordering while in SWMR write mode. Further, the metadata cache must provide the necessary support facilities.

VFD SWMR avoids involving the metadata cache clients in SWMR by taking periodic snapshots of the metadata at points when it is known to be consistent. These snapshots are then communicated via an out of HDF5 file store[1] to specialized VFDs in the reader processes. These reader VFDs intercept metadata read requests and satisfy them from the snapshots. This has the advantage of making SWMR transparent to all layers of the HDF5 library above the metadata cache – thus simplifying maintenance greatly, and allowing non-SWMR applications to avoid SWMR related overhead. Further, since the current state of HDF5 metadata is communicated outside the HDF5 file, the VFD SWMR approach opens the possibility of implementing SWMR on storage systems that do not support POSIX file I/O semantics. Finally, since the specialized VFD SWMR reader VFD is easily separated from the HDF5 library, SWMR support can be marketed as an add-on.

In the following, keep in mind that no version of SWMR makes guarantees about the state of raw data. They only guarantee is that the reader will see a consistent view of the metadata – not that the raw data read through the use of this metadata will have made it to file yet.

## 1.1 Outline of this Document

Section 2 is an updated version of the sketch design of VFD SWMR. It was written to propose the concept, and in this context, it is intended to provide a conceptual introduction. Note that it contains a number of oversimplifications, which are addressed in Section 3.

Section 3 is the design document to which VFD SWMR is implemented. It should fully define the function and design of all the code necessary to implement VFD SWMR. Note that this section will evolve as implementation proceeds, and unforeseen issues are addressed.

When VFD SWMR is fully implemented, section 4 will address code organization details needed for maintenance purposes.

Section 5 is the design document for the test suite needed to validate and maintain VFD SWMR. Initial versions will mostly list items to be tested. As implementation progresses, it should be updated to discuss the structure of the test code.

## 1.2 Update Post Phase II Award

Since the above introduction was written, we have implemented the initial proof of concept version of VFD SWMR and won the phase 2 contract to extend and rework it into a production version. We expect this task to be both easier and harder than one might expect.

Easier, because the initial implementation is remarkably complete, and requires only peripheral changes to convert it into a reasonable initial production version.

Harder, because we piggybacked on the existing SWMR test code, and wrote almost none of our own. While this was sufficient to demonstrate that the concept works, it is quite in-adequate for production purposes. Thus we must write a complete test suite (unit tests, integration tests, and performance tests) as one of our first orders of business.

While Section 2 of this document remains largely unchanged, Section 3 (VFD SWMR Design) has been reworked to clean up many of the short cuts and temporary solutions that were necessary to

[1] In the POSIX case, this is simply another file, separate from the HDF5 file and referred to later in this document as the metadata file. We use this circumlocution as matters are more complex in the NFS and object store cases.

implement the prototype within the time and budget allotted. Note, however, that the first production version must still co-exist with the existing SWMR implementation. Only when (and if) we decide to commit to VFD SWMR will we be able to begin the long and tedious task of removing the original implementation and simplify the code accordingly.

As of this writing (8/03/19), Section 4 (Implementation Details) remains empty. Work has started on Section 5 (Testing), and will continue as we specify and implement the test suite.

### 1.3 Update for the Friendly User Release

Since the previous update, we have addressed most of the above mentioned peripheral issues in the initial implementation, and developed a test suite that exercises most elements of the HDF5 file format in combination with VFD SWMR.

These tests have exposed a number of issues. While most of these issues have been dealt with, the following are outstanding:

- At present, the Virtual Data Sets (VDS) feature is not well integrated with VFD SWMR. While we have a work around that allowed us to test VDS for more fundamental issues, a proper solution is on hold pending availability of the original developer of VDS later this year.
- The current implementation of variable length fields in datasets is fundamentally incompatible with VFD SWMR due to the fact that variable length data is currently stored in global heaps accessed via the metadata cache. This is not as big a problem as it appears, as the current implementation of variable length data has very poor performance, and thus is not suitable for most SWMR applications. A re-implementation of variable length data is in the planning stage. While improved performance is the primary objective of the new implementation, it should also be compatible with VFD SWMR. Unfortunately, we have no ETA for this re-implementation.

  Variable length attributes (on groups or datasets) should work with VFD SWMR, but we don’t have a test for this yet.
- At present, the VFD SWMR implementation is hard coded to flush raw data at the end of each tick. While this has greatly simplified our initial test suite, it imposes extra overhead. Making this configurable, and adjusting our test suite accordingly is on our to-do list.

We have run initial performance tests, comparing elapsed time for a given sequence of writes to to a file opened for writing with VFD SWMR, to elapsed time for the same sequence of writes to a file opened in normal read / write mode. Results varied depending on the number and size of datasets, and the number of extensible dimensions.

Best results were obtained with small numbers (~5) of large data sets, where the VFD SWMR overhead was approximately zero.

Worst results were large numbers (~1000) of small data sets with two extensible dimensions, where overheads in the order of 100% were observed.

These results are preliminary, used synthetic loads, and are likely to change as we improve our performance tests, test on different systems, and optimize.

So far, we have not attempted similar performance tests on the VFD SWMR reader.

# 2 Conceptual Overview

Observe that HDF5 metadata must be in a consistent state at the beginning and end of API calls[2]. Thus we can safely make snapshots of HDF5 metadata at these points. To support a maximum latency from a given write to visibility of that write on the readers, on the writer side we must take snapshots on a regular basis, and on the reader side, we must check for updates regularly as well.

Let t be the desired maximum latency from write to visibility on the reader side. Define one tick to be t/3. With this definition in hand, consider the following outlines of the cycles of operation for the writer and readers.

Note that these outlines assume that all pieces of metadata are smaller than one page. While we should be able to get very close to this using the latest file format, this presumption is probably not attainable with practical page sizes. However, the assumption makes the cycle of operations easy to follow, and as shall be seen in section 3, the occasional exception can be handled easily as long the oversized pieces of metadata are not huge.

## 2.1 Writer Cycle of Operation

Presume that the file has been created with paged allocation, and that all pieces of metadata are no larger than a single page. Further suppose that we have modified the page buffer to track pages of metadata that have changed during the current tick, and to hold in memory any page that has been modified during the current tick.

Presume also that the API func enter / exit macros have been modified to check to see if the current tick has expired[3], and invoke the “writer_end_of_tick()” function if it has.

The writer_end_of_tick() function performs the following activities:

1. Flush the metadata cache to the page buffer.

2. Write all metadata pages that have been modified to the out of HDF5 file backing store. How this is done depends on whether the backing store is a POSIX file system, a NFS file system, or an object store. See below for discussions of each of these options.

3. Construct / update the index mapping the base addresses of all pages of metadata to locations in the out of HDF5 file backing store. Replace the old version of the index with the new version. How this is done again depends on the type of out of HDF5 file backing store used.

[2] This is a bit of an oversimplification, as some API calls allow the caller to specify callback routines, and these callback routines can invoke HDF5 library API calls. However, if we count API call entries and exits, and only consider initial entries and final exits of nested calls, the above statement is true.

[3] i.e. a tick (whatever period of time that may be) has passed since the current tick started.

4. Release space on the out of HDF5 file backing store that contains pages and/or indices that have been superseded more than max_lag ticks ago, where max_lag is user configurable and is least 10.

5. Make note of the start time of the new tick.

6. Resume normal processing.

Note that the writer_end_of_tick() processing does not require any writes to the HDF5 file proper. If the quantity of metadata is small enough to fit in the metadata cache and page buffer, there may be no metadata writes to the HDF5 file until file close.

Observe also that if an existing file is opened for VFD SWMR writing, there is no requirement that all metadata will be written to the out of HDF5 file backing store. Any metadata that has not been altered will remain in the file, and will be accessed normally by the reader. Note however, that if a pre-existing piece of metadata is modified, it may not be written to the HDF5 file for at least max_lag ticks lest a lagging reader receive a “message from the future”.

Due to this constraint, it is possible for a flush to require up to `max_lag` ticks to complete. While the flush raw data at end of tick option (discussed in section 3) should remove most if not all reasons to flush a file while it is open for VFD SWMR writing, this point should be kept in mind. Since the HDF5 file must be flushed as part of the close process, closing a file may take up to `max_lag` ticks as well.

### 2.1.1 Management of the Out of HDF5 File Backing Store

#### *2.1.1.1 POSIX*

In the case of a POSIX file system, pages of metadata are written to a metadata file[4] in such a fashion as to avoid overwriting any page of metadata that has been listed in the metadata page index in the last max_lag ticks. After all modified metadata pages are written to the metadata file, the old index is overwritten with the new version. In principle, this overwrite (along with the metadata page writes) should be atomic. However, past experience indicates that we should include checksums to allow the reader to detect torn writes, and re-try until the torn write completes.

Here the index maps base addresses of metadata pages in the HDF5 file to base addresses in the metadata file. Note that the metadata file need not be on the same physical file system as the HDF5 file proper – which avoids any file system contention between VFD SWMR related I/O and raw data I/O. If sufficient RAM is available, a small RAM disk would be ideal for the metadata file.

#### *2.1.1.2 NFS*

NFS guarantees neither write ordering nor atomic writes. However, from our cursory research, it does guarantee flush of all buffers on file close.

[4] The metadata file is sometime referred to as the “shadow file”. The terms are synonymous.

This suggests that, for NFS, the writer should construct a change list for the metadata file in a temporary file, close it, and then change its name to the next name in some well defined sequence of metadata file deltas.

The readers (or some helper process) would then check for an update every tick, and use all updates found to update a local copy of the metadata file.

To conserve file space, the update files could be deleted after max_lag ticks – although a large max_lag would be advisable to allow for network delays.

#### *2.1.1.3 Object Store*

The object store case is almost the same as the NFS case, with each metadata file change list being written to a new object.

## 2.2 Reader Cycle of Operation

As with the VFD SWMR writer, the page buffer must be used.

Presume that the VFD SWMR reader VFD is stacked on top of whatever VFD is used to read the HDF5 file proper, and intercepts all reads of metadata pages that are listed in the index. These reads are satisfied as directed by the index into the metadata file. The exact details of the SWMR VFD depend the details of the out of HDF5 file backing store – as before, POSIX, NFS, and objects stores are discussed individually below.

Presume also that the metadata cache has been modified so that it can invalidate all entries with base address within a specified range of addresses. Note that this may not be as easy as it sounds, as some metadata cache clients presume that metadata is loaded into the cache in a specific order – and thus may not react well to the eviction of randomly selected entries. The correct solution is to modify these cache clients to support refreshes of internal entries from file[5]. However, a workable interim solution is to simply evict the on disk data structure of which the target entry is part, and reload it if it is needed.[6]

Likewise, presume that the page buffer can evict all pages listed as having changed in the metadata file index.

Finally presume that the API func enter macros have been modified to check to see if the current tick has expired, and call the reader_start_new_tick() function if it has.

The reader_start_new_tick() function performs the following activities:

1. Direct the reader VFD to reload the index, and determine which pages have been modified since the last time the index was reloaded. For each modified page:
    - Evict the old version of the page from the page buffer.

[5] Strictly speaking, this violates the design objective of making SWMR transparent to all layers above the metadata cache.

[6] When last we discussed the issue, this is the solution that Quincey was planning to use for his implementation of full SWMR.

- Instruct the metadata cache to invalidate all entries located in the modified page.

2. Make note of the start time of the new tick, so that its end can be detected.

3. Resume normal processing.

### 2.2.1 Management of the Out of HDF5 File Backing Store

As indicated above, the details of the VFD SWMR reader VFD depend on the type of backing store used to store the metadata:

#### *2.2.1.1 POSIX*

In the case of POSIX file systems, when a page of metadata is requested by the page buffer, use the index to find the offset of the desired page in the metadata file, read the desired page, and pass it to the page buffer. When reading either the index or a metadata page, verify its checksum, and retry until the checksum is correct, or the maximum number of retries is exceeded.

Note that the index must provide a consistent view of the HDF5 file's metadata, as on the writer, the metadata cache was flushed to the page buffer before the index was created, and the tick ended at either the beginning or end of an API call. Further, no metadata page is overwritten until at least max_lag ticks have passed since the last time the page was mentioned in an index. Since the index is at most a little over 2 ticks old, since the page buffer is purged of any superseded pages each time a new index is loaded, and since any possibly superseded entries are likewise evicted from the metadata cache, this precludes any inconsistencies.

#### *2.2.1.2 NFS*

In the case of NFS file system, things are a bit more difficult, as there is no guarantee of write ordering. However, since NFS apparently guarantees full flush to backing store on close, the metadata file change list files discussed in 2.1.1.2 should be complete by the time they become visible to the reader SWMR VFDs.

When the SWMR VFD is directed to reload the index, it must query the NFS file system to see if any new metadata file change list files have become available. If any have, it must process these files in strict sequential order – if there is a gap, subsequent metadata file change list files must not be processed until the gap is filled.[7]

Once the next metadata file change list file has been identified, the SWMR VFD must read it and use it to update its local copy of the metadata file.[8] Once all actionable metadata file change list files have been processed, the SWMR VFD proceeds as per the POSIX case.

[7] While a gap need not halt processing on the reader, if it is not filled within `max_lag` ticks, the reader will likely perceive corruption in the metadata.

[8] While it should not be necessary, it would be prudent to include a checksum on the metadata file update files to ensure that NFS is behaving as expected.

Observe that if we create a separate process to monitor the NFS file system for metadata file update files, process them as they appear, and maintain a local copy of the metadata file, the NFS case resolves to the POSIX case from the perspective of the VFD SWMR reader. As this reduces the load on the SWMR VFD significantly, we should at least investigate this option when we get to NFS support.

#### *2.2.1.3 Object Store*

The object store case is almost the same as NFS, save that we may not have a way of ensuring atomic creation of metadata file change list objects. An obvious way of addressing this is to checksum the metadata file change list object and either retry or wait a tick if the checksum fails.

### 2.2.2 A Hidden Assumption

Our discussion of the VFD SWMR reader is not complete without discussing the hidden assumption that the reader that it will be able to complete each API call promptly – certainly within a tick.

This need not be the case on a heavily loaded system, where the scheduler and contention for access to the file system can introduce arbitrary delays.[9] In addition to breaking the real time requirement, if the delay exceeds max_lag ticks, it is possible that the reader will attempt to read a page of metadata from the metadata file that has been overwritten or deleted.

Given that an objective of VFD SWMR is to support real time access to data written to the HDF5 file, we could be forgiven for dismissing this problem on the grounds that the host system is not capable of meeting the specified real time constraint, and thus we have already failed, and we need not concern ourselves with secondary failures.

That said, not all users require true real time SWMR, and thus a brief discussion of possible solutions may be useful:

1. Increase tick length – thereby reducing the load on the host system.

2. Increase max_lag such that max_lag * (tick length) is greater than the maximum expected delay.

3. Modify the metadata cache entry load code to notice when more than max_lag-1 ticks have passed since the last time the index was loaded, and force a re-try of the API call if it has.

Options 1 & 2 are obvious, easy to implement, and should be supported. While option 3 is a possible solution[10], we reject it because re-introduces a great deal of SWMR specific code and complexity into

[9] It is also possible to construct HDF5 API calls which require arbitrarily large amounts of time to complete – for example very large reads or writes, particularly on complex selections. Fortunately, it should always be possible to avoid the problem by breaking such calls into an equivalent sequence of calls.

[10] And one that is used in the existing SWMR implementation

the library, for the purpose of supporting the user who is attempting to run VFD SWMR on marginal hardware with an insufficiently capable backing store. More to the point, a major impetus behind VFD SWMR is to minimize the amount of SWMR specific code in the HDF5 library proper, and thereby to simplify it and facilitate maintenance.

## 2.3 Maximum Delay from Write to Read

If we assume instantaneous file system response, and HDF5 API calls that are frequent relative to the tick frequency, the maximum delay from write to read with the above scheme should be a little more than two ticks – we add the remainder of the third tick to allow for I/O delays, delays between the end of a tick and the next API call on the writer, for writing metadata pages, and for constructing and writing indices.

This should be adequate assuming a POSIX file system that is not overloaded, and a tick size that is very large compared to the file system response time. Since the file system used for the metadata file need not be the same as that used for the HDF5 file proper[11], this latter constraint should be fairly easy to meet.

In contrast, this will likely not be the case with NFS and object stores unless the tick size is quite large (i.e. 10s of seconds, or more), since neither of these storage systems are designed for speed or to guarantee write ordering.

## 2.4 Parallel VFD SWMR

As should be obvious from the above cycles of operation, VFD SWMR is largely orthogonal to the normal operation of the HDF5 library. Thus, the only major additional requirement for using VFD SWMR with parallel computations is to enable the page buffer in parallel HDF5.

Given this, all that is needed to implement the VFD SWMR writer in parallel is to run the VFD SWMR writer code on one process – probably process 0. That process then writes modified metadata pages, and constructs and writes indices as per the serial case. Since all processes in parallel HDF5 see the same sequence of dirty metadata, this is sufficient.

With the extra processing on process 0, it may fall behind the other processes between sync points[12]. If this is an issue, additional sync points could be added. However, this will likely delay the overall computation. If a spare core is available, much of the VFD SWMR writer overhead could be offloaded to a thread. While this is probably a good idea in both the serial and parallel cases, it doesn't address the issue completely.

[11] System resources permitting, creating a small RAM disk for the metadata file would be ideal.

[12] In parallel HDF5, all processes perform all actions that modify metadata collectively, and thus see the same stream of dirty metadata. To allow the metadata caches to safely flush metadata entries, the metadata caches on all processes count the number of dirty bytes of metadata generated, and enter a sync point every n bytes, where n is user configurable. Once in the sync point, the process 0 metadata cache decides what entries to flush and then coordinates with the other metadata caches. This allows the metadata caches to flush and evict metadata without risking message from the past / future bugs.

The reader side of VFD SWMR is slightly more complex due to the difficulty of maintaining a consistent timer across multiple processes in a parallel computation. This makes it hard to ensure that all processes read the same index, and introduces the possibility of deadlocks.

To sidestep both of the issues, it will probably be necessary to require the reader application to run the `reader_start_new_tick()` function collectively from time to time. This allows us to designate a single process to read the index and broadcast it to the remaining processes – thereby ensuring a consistent view of the index. As the frequency of calls to `reader_start_new_tick()` will be under the control of the application, max_lag will have to be chosen to allow for the longest expected delay between calls to `reader_start_new_tick()`.

# 3 VFD SWMR Design

Note: This section of the VFD SWMR RFC is dated. Briefly, we have made a number of minor design changes to address issues encountered during our testing, and have not been as good at keeping this document up to date as we should be.

For efficient maintenance, this must be repaired before production release. However, until we do so, expect minor discrepancies between this section and the code.

While the above discussion of the cycle of operations for VFD SWMR should provide a good conceptual overview, as mentioned earlier, it contains one major oversimplification. Simply put, the HDF5 file format does not make it easy to set an upper bound on the size of pieces of metadata. Indeed, in older versions of the file format, it is possible to create arbitrarily large local heaps.

Fortunately, by requiring the latest file format for VFD SWMR, this issue can be largely tamed. However, even in this case and with default configuration, pieces of metadata can reach 64 KiB in size, and (under unusual circumstances) exceed it.

This means that while we can pick a metadata page size that is larger than the vast majority of pieces of metadata, we cannot guarantee that all metadata will fit in any given page size. Thus the implementation of VFD SWMR must be able to handle this eventuality.

Fortunately, when paged allocation is enabled, if space for a piece of metadata larger than one page is requested, the free space manager allocates the smallest integral number of adjacent pages required, allocates the requested space starting at the beginning of this sequence of pages, and does not allocate space from the page fragment at the end.

This means that if a piece of metadata larger than one page is flushed from the metadata cache to the page buffer either during or at the end of a tick, it is sufficient for the page buffer to retain a copy, write it to the out of HDF5 file backing store, and include it in the index in the usual end of tick processing for the VFD SWMR writer. Further, since we know that any space between the end of the larger than one page piece of metadata and the end of the last page is un-allocated, we need not concern ourselves with this file space.[13]

[13] Unfortunately, testing has revealed two cases in which this is not true. Both the fixed array and extensible array indexing structures allocate large blocks of file space, and sub-allocate metadata entries out of it. This in turn creates the situation in which a metadata cache entry can be of size

Note however, that having to deal with metadata entries larger than one page does complicate free space management in the metadata file that is maintained in the POSIX case (and likely the NFS and Object store cases as well).

## 3.1 API Additions

### 3.1.1 FAPL Additions

The current SWMR implementation allows the user to shift to SWMR writer mode after the file has been opened. As I understand it, the initial SWMR implementation did not support creation of groups and datasets in a SWMR safe way, and thus it was necessary to create all needed groups and datasets before allowing the file to be read by SWMR readers. I gather that this limitation has been addressed as part of Quincey's "Full SWMR" project, which is an extension of the current SWMR design.

VFD SWMR doesn't have this issue, and thus we can simply specify VFD SWMR on file open or create. We will do this with the new FAPL (File Access Property List) entry H5F_VFD_SWMR_CONFIG.

The signatures for the calls for getting and setting this property are:

```
 herr_t H5Pset_vfd_swmr_config(hid_t plist_id, H5F_vfd_swmr_config_t *config_ptr);

 herr_t H5Pget_vfd_swmr_config(hid_t plist_id, H5F_vfd_swrm_config_t *config_ptr);
```

Where H5F_vfd_swmr_config_t is defined as follows:

```
/******************************************************************************
 *
 * struct H5F_vfd_swmr_config_t
 *
 * Instances of H5F_vfd_swmr_config_t are used by VFD SWMR writers and readers
 * to pass necessary configuration data to the HDF5 library on file open (or
 * creation, in the case of writers).
 *
 * Given that the VFD SWMR configuration FAPL property is set, the writer field,
 * (discussed below) must be consistent with the flags passed in to H5Fopen()
 * (either H5F_ACC_RDWR for the VFD SWMR writer, or H5F_ACC_RDONLY for the VFD
 * SWMR readers).
 *
 * If H5Fcreate() is used and the VFD SWMR FAPL property is set, the file will
 * be opened as a VFD SWMR writer (and the writer field must be set to TRUE).
 *
 * It is the user's responsibility to ensure that there is exactly one VFD SWMR
 * writer for any file that is accessed as a VFD SWMR file.
 *
 * Further, the user must ensure that the VFD SWMR FAPL entries on the writer
 * and reader(s) are consistent – i.e. tick_len, max_lag, md_pages_reserved, and
 * md_file_path must match.
 *
 * The fields of H5F_vfd_swmr_config are discussed below:
 *
```

larger than one page, and not be page aligned. While this permits a minor reduction in the on disk size of these indices, as shall be seen, it complicates VFD SWMR support in the page buffer.

```
* version: Integer field indicating the version of the H5F_vfd_swmr_config
*         structure used.  This field must always be set to a known version
*         number.  The most recent version of the structure will always be
*         H5F__CURR_VFD_SWMR_CONFIG_VERSION.
*
* tick_len: is an integer field containing the length of a tick in tenths of
*         a second.  If tick_len is zero, end of tick processing may only be
*         triggered manually via the H5Fvfd_swrm_end_tick() function.
*
* max_lag is an integer field indicating the maximum expected lag (in ticks)
*         between the writer and the readers.  This value must be at least 3,
*         with 10 being the recommended minimum value.
*
* writer: Boolean flag indicating whether the file opened with this FAPL entry
*         will be opened R/W. (i.e. as a VFD SWMR writer)
*
* flush_raw_data: Boolean flag indicating whether raw data should be flushed
*         as part of end of end of tick processing.  If set to TRUE, raw
*         data will be flushed and thus be consistent with the metadata file.
*         However, this will also greatly increase end of tick I/O, and will
*         likely break any real time guarantees unless a very large tick_len
*         is selected.
*
* md_pages_reserved:  Integer field indicating the number of pages reserved
*         at the head of the metadata file.  This value must be greater than
*         or equal to 1.
*
*         When the metadata file is created, the specified number of pages is
*         reserved at the head of the metadata file.  In the current
*         implementation, the size of the metadata file header plus the
*         index is limited to this size.
*
*         Further, in the POSIX case, when readers check for an updated index,
*         this check will start with a read of md_pages_reserved pages from
*         the head of the metadata file.
*
* md_file_path: In the POSIX case, this field contains the path of the
*         metadata file.
*
*         In NFS, it contains the path and base name of the metadata file
*         updater files.
*
*         For an object store, it contains the base URL for the objects used
*         to store metadata file updater objects.
*
* log_file_path: path to log file.  If defined, this path should be unique to
*         each process.  If this field contains the empty string, a log file
*         will not be created.
*
* pb_expansion_threshold: During a tick, the page buffer must expand as
*         necessary to retain copies of all modified metadata pages and multi-
*         page metadata entries.  This field allows the user to specify a
*         threshold on page buffer size, which if exceeded, will trigger an
*         early end of tick.  Note that this is not a limit on the maximum
*         page buffer size, as the metadata cache is flushed as part of end
*         of tick processing.
*
*         The pb_expansion_threshold is an integer which must be in the range
*         [0, 100].
```

```
 *
 *         If the pb_ezpansion_threshold is 0, the feature is disabled.
 *
 *         For all other values, the page buffer size is multiplied by the
 *         pb_expansion_threshold.  If this value is exceeded, an early end
 *         of tick is triggered.
 *
 ******************************************************************************/

#define H5F__CURR_VFD_SWMR_CONFIG_VERSION 1

typedef struct H5F_vfd_swmr_config_t {

    int32 version;
    int32 tick_len;
    int32 max_lag;
    hbool_t writer;
    hbool_t flush_raw_data;
    int32 md_pages_reserved;
    char[MAX_PATH+1] md_file_path;
    char[MAX_PATH+1] log_file_path;
    int32_t pb_expansion_threshold;

} H5F_vfd_swmr_config_t;
```

Note that if the VFD SWMR configuration is set in the FAPL, the file open / create must fail if any of the following conditions hold:

- The call used to open or create the file doesn't match the value of the writer field in the VFD SWMR FAPL entry,
- Paged allocation was not specified in the FCPL (File Creation Property List) on file creation, or
- Page buffering was not enabled in the FAPL.

### 3.1.2 End Tick API Call

The H5Fvfd_swmr_end_tick() API call exists to allow the user to trigger end of tick processing on either the VFD SWMR reader or writer. The signature of the API call is given below:

```
     herr_t H5Fvfd_swmr_end_tick(hid_t file_id);
```

This call is necessary if the user elects to manage ticks manually, and may also be used by the writer to propagate changes early if it knows that either the HDF5 library will not be called for an extended period, or that no further changes will be made for a while.

This function must fail if the target file is not opened with VFD SWMR. Similarly, the function must fail if it is called while end of tick is disabled (see section 3.1.3 below).

Note that this function must be implemented in such a way that the end of tick processing will only be executed once in cases where end of tick would otherwise by triggered by the FUNC ENTER/EXIT macros (see below).

### 3.1.3 Enable / Disable End of Tick Call

It will sometimes be useful to allow the writer or reader to briefly delay end of tick processing so that it does not fall in the middle of a sequence of operations that are best viewed as atomic. The

H5Fvfd_swmr_disable_end_of_tick() and H5Fvfd_swmr_enable_end_of_tick() calls exist to support this. The signatures of these API calls are given below:

```
herr_t H5Fvfd_swmr_disable_end_of_tick(hid_t file_id)
herr_t H5Fvfd_swmr_enable_end_of_tick(hid_t file_id)
```

If a call to H5Fvfd_swmr_disable_end_of_tick()is made for a given file ID, the end of tick function will not be called until the matching call to H5Fvfd_swmr_enable_end_of_tick()is made. Note that in addition to re-enabling tests for end of tick on the target file, the enable end of tick must check to see if the tick has expired, and trigger end of tick processing if it has.

Note that these calls should only effect the specified file, and that it is an error to attempt to disable end of tick processing for a file for which it is already disabled, and vice versa.

The user should be cautioned to disable end of tick processing only for periods of time that are short in comparison to the current tick length.

It is also an error to call H5Fvfd_swmr_end_tick() while end of tick processing is disabled.

## 3.2 Modifications to Existing Data Structures and New Data Structures

### 3.2.1 Additions to H5F_file_t

When the HDF5 file is opened (or created) with VFD SWMR, it is necessary to store configuration data, time of the end of the current tick, etc. some place convenient that is associated with the target file. Add the following fields to H5F_file_t:

```
    /* VFD SWMR info */
    hbool_t vfd_swmr;        /* Boolean flag indicating whether the file has */
                             /* been opened with VFD SWMR configured.  All   */
                             /* other fields in this section are undefined   */
                             /* if this field is FALSE                       */
    hbool_t vfd_swrm_writer; /* Boolean flag that is set to TRUE iff this is */
                             /* is the VFD SWMR writer.                      */
    H5FD_vfd_swmr_idx_entry_t * md_file_index; /* Pointer to a dynamically   */
                             /* allocated array of instances of              */
                             /* H5FD_vfd_swmr_idx_entry_t.
    uint64_t tick_num;       /* Number of the current tick.  This field is   */
                             /* initialized to zero, and incremented at the  */
                             /* end of each tick.                            */
    struct timespec end_of_tick; /* End time of the current tick.  This      */
                             /* value is initialized at file open, and       */
                             /* updated at the end of each tick.             */
    int vfd_swmr_md_file;    /* In the posix case, vfd_swmr_md_file is the   */
                             /* file descriptor of the metadata file, or -1  */
                             /* if the metadata file is not currently open.  */
                             /* This field is not used and is set to -1 in   */
                             /* the NFS and object store cases.              */
    int vfd_swmr_log_file;   /* File descriptor of the VFD SWMR log file if  */
                             /* defined and open.  Otherwise it is set to -1.*/
    H5F_vfd_swmr_config_t vfd_swmr_config; /* copy of the vfd swmr           */
                             /* configuration from the FAPL use to open the  */
                             /* file.                                        */
```

### 3.2.2 New Global Data Structures

Some API calls don’t reference files, and for those that do, there is no guarantee that the supplied file ID will reference the VFD SWMR file. Thus, to allow the API FUNC ENTER/EXIT macros to detect the end of tick, and trigger end of tick processing on the appropriate file, we must make it possible for the macros to detect if a file is opened in VFD SWMR writer or reader mode, and determine when the current tick should end.

In principle, there can be an arbitrary number of files opened in an arbitrary mix of VFD SWMR writer, VFD SWMR reader, regular R/W, or regular R/O modes. Thus we must maintain a queue of tick expiration times decorated with pointers to the associated instances of H5F_file_t and Booleans indicating either writer or reader mode.

Call this queue the EOT queue, and implement it as a doubly linked list of instances of the `H5F_vfd_swmr_eot_queue_entry_t` structure defined below:

```
/******************************************************************************
 *
 * struct H5F_vfd_swmr_eot_queue_entry_t
 *
 * Instance of this structure are used to maintain an end of tick time sorted
 * list of files opened in either VFD SWMR write or VFD SWMR read mode.  Each
 * structure contains all information required to determine whether the end of
 * tick has arrived for the specified file, and to initiate end of tick
 * processing if it has.
 *
 * Since this list is maintained in increasing end of tick time order, only the
 * first item need be inspected if its end of tick time has not expired.
 *
 * The fields of H5F_vfd_swmr_eot_queue_entry_t are discussed below:
 *
 * vfd_swmr_file: Pointer to the instance of H5F_file_t containing the shared
 *         fields of the associated file that has been opened in VFD SWMR mode
 *
 * vfd_swrm_writer:  Boolean flag that is set to TRUE if the associated file
 *         has been opened in VFD SWMR writer mode, and FALSE if it has been
 *         opened in VFD SWMR reader mode.
 *
 * tick_num: Number of the current tick of the target file.
 *
 * end_of_tick: Expiration time of the current tick of the target file.
 *
 * next: Pointer to the next element in the end of tick queue, or NULL if there
 *         is no next entry.  Note that if next is not NULL, next->end_of_tick
 *         must be greater than or equal to end_of_tick.
 *
 * prev: Pointer to the previous element in the end of tick queue, or NULL if
 *         there is no previous entry.  Note that if prev is not NULL,
 *         prev->end_of_tick must be less than or equal to end_of_tick.
 *
 ******************************************************************************/

typedef struct H5F_vfd_swmr_eot_queue_entry_t {

    hbool_t vfd_swrm_writer;
    uint64_t tick_num;

    struct timespec end_of_tick;

    H5F_file_t *vfd_swmr_file;

    H5F_vfd_swmr_eot_queue_entry_t * next;

    H5F_vfd_swmr_eot_queue_entry_t * prev;

} H5F_vfd_swmr_eot_queue_entry_t;
```

Observe that there will be exactly one instance of `H5F_vfd_swmr_eot_queue_entry_t` for each file opened in VFD SWMR mode. This has two implications:

1. The same instance of `H5F_vfd_swmr_eot_queue_entry_t` can be re-used each tick, thus avoiding the overhead of repeated allocation and de-allocation.
2. Assuming (as seems likely) that there will be neither large numbers of files opened in VFD SWMR mode in a single process, nor large variations in tick length between such files, implementing the queue as a doubly linked list should be reasonably efficient.

The head and tail of the end of tick queue will be maintained in the global variables:

```
H5F_vfd_swmr_eot_queue_entry_t * vfd_swmr_eot_queue_head;
H5F_vfd_swmr_eot_queue_entry_t * vfd_swmr_eot_queue_tail;
```

To minimize overhead, the end of tick and whether the target file is a VFD SWMR writer must also be cached in globals:

```
hbool_t vfd_swmr_writer;
struct timespec end_of_tick;
```

Observe that it is sufficient to test `(vfd_swmr_eot_queue_head != NULL)` to determine whether there is a file opened in VFD SWMR mode.

When a file is opened in VFD SWMR mode, an instance of `H5F_vfd_swmr_eot_queue_entry_t` must be allocated, initialized, and inserted on the EOT queue in the appropriate location. Do this by starting at the tail of the queue, and inserting the entry after the first entry encountered such that `end_of_tick` less than or equal to that of the new entry, or at the head of the queue if no such entry exists. In this latter case, the global variables `vfd_swmr_writer` and `end_of_tick` must also be set equal to the fields of the same name in the new instance.

Observe that this insertion algorithm ensures that the EOT queue is sorted in `end_of_tick` order.

When a file that has been opened in VFD SWMR mode is closed, the above procedure must be reversed. The associated instance of `H5F_vfd_swmr_eot_queue_entry_t` must be removed from the EOT queue and discarded. Further, if the instance was at the head of the queue, the global variables `vfd_swmr_writer` and `end_of_tick` must be set equal to the fields of the same name of the next instance of the queue, if such an instance exists. If no such instance exists, no action is required, as the `vfd_swmr_eot_queue_head` will be NULL, indicating that there are no files opened in VFD SWMR mode.

### 3.2.3 Internal Representation of the Metadata File Index

Arrays of the `H5FD_vfd_swmr_idx_entry_t` structure are used to represent the metadata file index internally, both for the writer and reader.

The definition of this structure is given below:

```
/******************************************************************************
 *
 * struct H5FD_vfd_swmr_idx_entry_t
 *
 * Indices into the VFD SWMR metadata file are maintained in arrays of
 * instances of H5FD_vfd_swmr_idx_entry_t.
 *
 * The fields of H5FD_vfd_swmr_idx_entry_t are discussed below:
 *
 * hdf5_page_offset: Unsigned 64-bit value containing the base address of the
 *         metadata page, or multi page metadata entry in the HDF5 file IN
 *         PAGES.  To obtain byte offset, multiply this value by the page size.
 *
 *         WARNING: This value may be stored in a smaller field in the
 *         metadata file.  When this is done, be sure to make the appropriate
 *         conversions.
 *
 * md_file_page_offset: Unsigned 64-bit value containing the base address of
 *         the metadata page, or multi page metadata entry in the metadata file
 *         IN PAGES.  To obtain byte offset, multiply this value by the page
 *         size.
 *
 *         WARNING: This value may be stored in a smaller field in the
 *         metadata file.  When this is done, be sure to make the appropriate
 *         conversions.
 *
 * length: Unsigned 32-bit value containing the length of the metadata page or
 *         multi page metadata entry IN BYTES.  If this is a metadata page,
 *         the length must equal the page size.  If this is an individual multi
 *         page cache entry, the length must be greater than the page size, but
 *         need not be a multiple of the page size
 *
 * checksum:  Checksum of the metadata page or multi-page metadata entry
 *         referenced by this index entry.  On the writer side, this value
 *         is undefined until the referenced entry has been written to the
 *         metadata file.
 *
 * entry_ptr:  Used by the VFD SWMR writer only.  For the reader, this field
 *         should always be NULL.
 *
 *         If the referenced metadata page or multi-page metadata cache entry
 *         was modified in the current tick, this field points to a buffer in
 *         the page buffer containing its value.
 *
 *         This pointer is used by the metadata file creation / update code to
 *         access the metadata pages / multi-page metadata entries so that their
 *         current values can be copied into the metadata file.  After this copy,
 *         the entry_ptr field should be set to NULL.
 *
 * tick_of_last_change: Number of the last tick in which this index entry was
 *         changed.  This field is only used by the VFD SWMR writer.  For
 *         readers, it will always be set to 0.
 *
 * clean:  Boolean field used only by the writer.  It is set to TRUE whenever
 *         the referenced metadata page or multi-page metadata cache entry is
 *         written to the HDF5 file, and FALSE, whenever it is marked dirty in
 *         in the page buffer
 *
 *         For the reader, it should always be set to TRUE.
 *
 * tick_of_last_flush: Number of the tick in which this entry was last written
 *         to the HDF5 file, or zero if it has never been flushed.
 *
 *         This field is used only by the VFD SWMR writer.  For the reader, it
 *         should always be zero.
 *
 * delayed_flush:  If the flush of the referenced metadata page or multi-page
 *         metadata cache entry must be delayed, the earliest tick in which it
 *         may be flushed, or zero if there is no such constraint.
 *
 *         This field is used only by the VFD SWMR writer.
 *
 *         Flushes must be delayed whenever an entry:
 *
 *           1) appears in the HDF5 file, and
 *
 *           2) is newly inserted into the metadata file.
 *
 *         This is necessary, as if the above conditions occur, the write of
 *         the modified page or multi-page metadata cache entry must be delayed
 *         for at least max_lag ticks as otherwise a reader using an earlier
 *         version of the index may read the target from the HDF5 file and get
 *         a message from the future.
 *
 *         The above situation can occur when VFD SWMR is used on existing file,
 *         or after a flush.
 *
 * moved_to_HDF5_file: Boolean flag that is set to TRUE iff the entry referenced
 *         is clean, was written to the HDF5 file more than max_lag ticks ago,
 *         and is about to be removed from the index.
 *
 *******************************************************************************/

typedef struct H5FD_vfd_swmr_idx_entry_t {

    uint64_t hdf5_page_offset;
    uint64_t md_file_page_offset;
    uint32_t length;
    uint32_t checksum;
    void *   entry_ptr;
    uint64_t tick_of_last_change;
    hbool_t  clean;
    uint64_t tick_of_last_flush;
    uint64_t delayed_flush;
    hbool_t moved_to_HDF5_file;

} H5FD_vfd_swmr_idx_entry_t;
```

The VFD SWMR writer maintains an array of `H5FD_vfd_swmr_idx_entry_t`, and passes it to the metadata file writer code to handle the details of creating / updating the metadata file.

Similarly, the VFD SWMR reader VFD stores its internal representation of the index in an array of H5FD_vfd_swmr_idx_entry_t, and supplies copies of this array to the reader end of tick processing code on request. Finally, as discussed later in this document, the reader must also retain a copy of the previous version of the index to direct metadata cache updates when a new version of the index is read by the VFD SWMR reader VFD.

## 3.3 API FUNC ENTER / EXIT Macro Modifications

When VFD SWMR is enabled, on each API call, the HDF5 library must test to see if a tick has expired, and trigger the appropriate processing if it has. At the HDF5 library already has the API FUNC ENTER / EXIT macros that are executed on API function entry and exit, this is the obvious place to insert this check.

For the VFD SWMR writer case, the check for end of tick must be performed on both API call entry and exit so as to maximize the regularity with which the metadata file is updated. Since the VFD SWMR readers will not see any changes to the metadata file until the next API call entry, there is no need to check on API call exit[14].

To this end, the API FUNC ENTER / EXIT macros must be modified as follows.

1. Test to see if VFD SWMR is enabled (i.e. if vfd_swmr_eot_queue_head is not NULL). If it is disabled, we are done. Otherwise, make note of the current value of vfd_swmr_eot_queue_head and proceed to 2.
2. For the API FUNC EXIT macros, test to see if we are the VFD SWMR writer (i.e. if vfd_swmr_writer is TRUE). If we are not, we are done.
3. Test to see if the tick has expired. If it hasn't, we are done.
4. If vfd_swrm_writer is TRUE, call the writer end of tick function. Otherwise, call the reader end of tick function.
5. If we get this far, it is possible that there are additional files open in VFD SWMR mode whose current ticks have expired. If vfd_swmr_eot_queue_head is not NULL, and not equal to the value noted in step 1, goto step 2. Otherwise we are done.

Note that end of tick function must:

1. Remove the associated H5F_vfd_swmr_eot_queue_entry_t from the EOT queue,
2. Update it,
3. Reinsert it in end_of_tick order as discussed in section 3.2.2 above, and
4. Set the vfd_swmr_writer and end_of_tick globals to the values of the fields of the same name in the instance of H5F_vfd_swmr_eot_queue_entry_t at the head of the EOT queue.

[14] Note that due to callbacks from HDF5 into the host program, HDF5 may receive additional API calls before the original API call exits. This is a problem, as we may not be in a stable state when one of the additional API calls is made. Handle this by creating an API call depth counter, incrementing on API FUNC ENTER, decrementing on API FUNC EXIT, and only testing for end of tick when the depth counter is zero.

Observe that above algorithm allows an expired writer end of tick to be masked by a reader end of tick that precedes it in the EOT queue at API function exit. Note that this will happen only occasionally, and when it does, it will delay the writer EOT only until the next API function entry. Since we don't see many plausible use cases for a single process simultaneously opening files in both VFD SWMR writer and VFD SWMR reader mode, the added overhead required to address this issue does not seem warranted. This judgment may or may not be correct, and should be documented in the appropriate header comment.

### 3.3.1 The Time Function

Since we much check for end of tick on every API call entry and exit, this test must be done cheaply. For the first cut, we will use the system call *clock_gettime()*[15] to retrieve the current time of the specified clock:

*clock_gettime(clockid_t clk_id, struct timespec *curr_time);*

- Use CLOCK_MONTONIC for *clk_id* as this is available across Linux, Solaris and Mac
    - Note the following:
        - Certain clocks like CLOCK_MONOTONIC_COARSE is not chosen because it is Linux-specific
        - CLOCK_MONOTIONIC is the alternate name for CLOCK_HIGHRES on Solaris
        - *clock_gettime()* is not defined before macOS 10.12
- *curr_time* is:

```
struct timespec {
    time_t  tv_sec;     /* seconds */
    long    tv_nsec;    /* nanoseconds */
}
```

However, if this call proves too expensive, we will have to look at other options. Note also that we will eventually have to get this working on Windows as well.

### 3.3.2 The Function to Test for End of Tick

Pseudo code for the function to test for end of tick is outlined below:

```
vfd_swmr_test_for_end_of_tick(hbool_t reader_exit)
{
    H5F_vfd_swmr_eot_queue_entry_t init_eot_queue_head = NULL;

    if (vfd_swmr_eot_queue_head != NULL )
    {
        init_eot_queue_head = vfd_swmr_eot_queue_head;

        do {
            // get current time via
            // clock_gettime(CLOCK_MONOTONIC, curr_time);
```

[15] Or possibly `gettimeofday()`.

```
            if ( ( curr_time.tv_sec >= end_of_tick.tv_sec ) &&
                 ( curr_time.tv_nsec >= end_of_tick.tv_nsec ) )
            {
                if ( vfd_swmr_writer )
                    // call writer end of tick function
                else if ( ! reader_exit )
                    // call reader end of tick function
                else
                    // break out of the do-while loop.  This is
                    // where it is possible that writer end of tick
                    // may be masked by a reader end of tick.
            } else {
                // break out of do-while loop.
            }
        } while ( ( vfd_swmr_eot_queue_head != NULL ) &&
                  ( vfd_swmr_eot_queue_head != init_eot_queue_head ) );
    }
}
```

To avoid function call overhead, this function should be implemented as a macro.

Note that the above pseudo code presumes that instances of `H5F_vfd_swmr_eot_queue_entry_t` are recycled, and that end of tick functions update the EOT queue and the associated global variables as discussed in section 3.3 above.

### 3.3.3 The API Entry Macro

We will invoke *vfd_swmr_test_for_end_of_tick()* towards the end of the FUNC_ENTER_API macro:

```
FUNC_ENTER_API_COMMON
FUNC_ENTER_API_INIT(err);
H5E_clear_stack(NULL);
Call vfd_swmr_test_for_end_of_tick(FALSE)
{
```

There are other forms of the API entry macros:

- FUNC_ENTER_API_NOCLEAR
  - This macro is used for API functions that should not clear the error stack like *H5Eprint* and *H5Ewalk*
  - We will invoke *vfd_swmr_test_for_end_of_tick()* in a similar way:

    ```
    FUNC_ENTER_API_COMMON
    FUNC_ENTER_API_INIT(err);
    Call vfd_swmr_test_for_end_tick(FALSE)
    {
    ```

- FUNC_ENTER_API_NOINIT
  - This macro is used for API functions that do not perform _any_ initialization of the library or an interface, just perform tracing etc.  Examples are: H5allocate_memory, H5is_library_threadsafe, etc.

  - No change
- FUNC_ENTER_API_NOINIT_NOERR_NOFS
  - This macro is used for API functions that do not perform _any_ initialization of the library or an interface or push themselves on the function stack, just perform tracing, etc.  Examples are: H5close, H5check_version, etc.
  - No change

### 3.3.4 The API Exit Macro

We will invoke *vfd_swmr_test_for_end_of_tick()* at the beginning of the FUNC_LEAVE_API macro:

```
Call vfd_swmr_test_for_end_of_tick(!vfd_swmr_writer)
FUNC_LEAVE_API_COMMON(ret_value);
(void)H5CX_pop();
H5_POP_FUNC
if(err_occurred)
      (void)H5E_dump_api_stack(TRUE);
FUNC_LEAVE_API_THREADSAFE
return(ret_value);
```

There are other forms of the API exit macros:

- FUNC_LEAVE_API_NOINIT
  - This macro is used to match the FUNC_ENTER_API_NOINIT macro
  - No change
- FUNC_LEAVE_API_NOFS
  - This macro is used to match the FUNC_ENTER_API_NOINIT_NOERR_NOFS macro
  - No change

## 3.4 Page Buffer Re-Design

The functional requirements for the page buffer in VFD SWMR are listed below:

1. Retain copies of all metadata pages modified during the current tick. Copies may be clean or dirty (but see 3 below).
2. Retain copies of all multi-page metadata writes during the last tick. Copies may be clean or dirty (but see 3 below).
3. If a page of metadata or a multi-page metadata entry exists in the hdf5 file, and is not mentioned in the metadata file index, and is then written to the page buffer, it must not be flushed to the HDF5 file for at least max_lag ticks.  This is necessary, as metadata reads not listed in the metadata file are satisfied from the HDF5 file.  Thus writing the entry to the HDF5 file before max_lag ticks have elapsed may result in a lagging reader receiving a message from the future – which will be indistinguishable from file corruption.

This situation can arise if an existing file is opened VFD SWMR write, or if a file that is created in VFD SWMR write mode is flushed.

Thus the page buffer must provide mechanisms for:

   a. Determining if a page or multi-page metadata entry has been read from the HDF5 file since either file open of the last flush.

   b. If it has, there must be a mechanism for delaying its write to the HDF5 file for at least max_lag ticks since since it appeared in the metadata file index.

NOTE: Due to this latter requirement, a flush of the HDF5 file must perform all possible flushes, and then repeatedly sleep for a tick and try again until all write delays are satisfied.

4. Provide a convenient mechanism for locating all metadata pages and multi-page pieces of metadata that have been modified in the current tick.

5. The page buffer must track the total size of the pages and/or multi-page metadata entries modified or inserted in the current tick. There must also be a facility for triggering the end of tick early if this size exceeds a user provided limit.

Observe that these functional requirements necessitate a page buffer that can handle variable size entries, and that can expand and contract as needed. Unfortunately, the pre-existing page buffer supports neither of these facilities, and seems architecturally un-suited to the task.

The initial thought was that the metadata cache supports most of the desired functionality, and thus should be easily extensible to provide the missing features. However, during the implementation of the initial prototype, time pressure and the resulting need to avoid changes to the page buffer test code drove the decision to implement a new page buffer.

Extending the metadata cache so that it can also perform the roll of the page buffer is still an option. However, the only reason for doing so is to minimize maintenance costs, and it is not clear that it makes economic sense to do so. In any case, there is little point in considering this until VFD SWMR is fully implemented.

In the interim, the new page buffer exists, and appears to be functional[16]. The remainder of this section documents the new page buffer internals.

### 3.4.1 Metadata Allocation and I/O Complications Requiring Changes to the Page Buffer

The original re-write of the page buffer assumed the following invariants on metadata space allocation and I/O:

1. The file space for each metadata entry is allocated individually. In the context of paged allocation this implies that:

   a. If a piece of metadata is smaller than a page, it does not cross page boundaries.

[16] There is an occasional assertion failure that appears in the page buffer during existing VFD SWMR regression tests. It has not been investigated, as it does not appear to bare on the question of the viability of the VFD SWMR design concept. Needless to say, this issue must be addressed as part of phase 2.

   b. If it is of page size or larger, its base address is page aligned.

2. Metadata entries are read and written atomically – that is to say that the entire on disk representation of a metadata entry is read and written in a single I/O call.

These invariants were assumed in the initial re-implementation of the page buffer and the metadata file. In the context of the page buffer, they were mostly used for sanity checking, and to permit easy identification of multi-page metadata entries. In the metadata file management code, they are used both for sanity checking and to allow more efficient management of multi-page metadata entries.

While the above invariants have been true historically, the recent extensible array (H5EA) and fixed array (H5FA) on disk data structures violate the first invariant[17]. While there appears to be no developer level documentation of either the fixed array or the extensible array structures for indexing chunked datasets, they are discussed in some detail in "HDF5 Single-Writer/Multiple-Reader Feature Design and Semantics" (https://support.hdfgroup.org/HDF5/docNewFeatures/SWMR/Design-HDF5-SWMR-20130629.v5.2.pdf).

Between the above reference and a scan of the source code, it appears that these chunked dataset indexing data structures construct large on disk tables of chunk addresses and lengths. Further, to minimize footprint in the metadata cache, these tables are broken into "pages" of size unrelated to the page size used in paged allocation. Each of these "pages" is managed as a separate metadata entry. While this is not a problem in itself, blocks of these "pages" are allocated in a single H5MF_alloc() call. While this simplifies the fixed and extensible array on disk data structures by making it possible to compute the base address of each page from the base address of the table, in the context of paged allocation, it makes it possible for these "pages" to either:

- Be of size less than a paged allocation page and cross a page boundary, or
- Be of size greater than or equal to a paged allocation page, and not have a base address that is a multiple of the paged allocation page size.

The correct solution to this problem is to modify the fixed and extensible array on disk structures to conform with the first invariant. However, this requires a file format change that can't be applied until HDF5 1.14. Further, we must either handle the current versions for the indefinite future, or detect them and throw an error if the user attempts to open a data set that uses one of them.

#### *3.4.1.1 Outline of Needed Modifications to Fixed and Extensible Array Data Structures*

While the fixed and extensible array on disk data structures could be modified to allocate their "pages" individually, and maintain the necessary indices of "page" addresses, this may not be necessary, as it should be possible to modify the existing scheme to conform to the first invariant by making it aware of the paged allocation page size, and inserting padding as necessary to conform with the invariant. Indeed, if I apprehend the situation correctly, the extensible array on disk data structure already does this as long as:

- The paged allocation page size is a power of two, and
- The file offset and length fields are both the same length, and that length is a power of two.

[17] This point was recognized in late 2019, with this update to the RFC added in January 2020.

As a page size of 4096 bytes was used in the prototype tests, and since the default size of offsets and lengths in 8 bytes, this may explain why the issue was not detected earlier.

Unfortunately, the fixed array on disk data structure doesn't make things this easy – although allocating the header and the table of chunk addresses and lengths separately should make the problem essentially identical to the extensible array case.

#### *3.4.1.2 Supporting the Existing Versions of the Fixed and Extensible Array Data Structures*

Be all this as it may, we still have to modify the page buffer and metadata file to handle fixed and extensible array "pages" that don't conform to the first invariant.

In the page buffer, we simply have to modify the metadata I/O code to permit metadata writes that are either less than a page in length and cross page boundaries, or that are of length greater than or equal to the page size and don't start on a page boundary. We already do this for raw data, so the major issues should be to break such I/O's into two or three pieces that respect the first invariant (i.e. either start on a page boundary or don't cross a page boundary), and then apply them as before (Note that both leading and trailing page fragments must be treated as individual I/O operations). This code should also contain sanity checking to verify that the metadata being read or written is either a fixed or extensible array "page", and throw an assertion if it isn't. It would also be useful to log such I/O operations so we can see how common they are.

If we do this, no changes should be necessary in the metadata file and its supporting code. While breaking the I/O operations up will increase the size of the index, it shouldn't increase by more than a factor of three – and that should be rare.

It will also be necessary to review the reader EOT code to verify that code that assumes the first invariant doesn't cause us to skip evicting some entries. We will have to review the code to be sure, but I think any changes will be minor.

### 3.4.2 Architectural Overview

Architecturally, the new page buffer is similar to the metadata cache.

Entries are indexed with a hash table with chaining. Like the metadata cache, the hash table size must be a power of two. This permits a very fast hash function on the page offset (page base address / page size), that simply bit ands the page address with the hash table size – 1. This unusual design decision is based on the observation that if the principle of locality holds, collisions between hot pages are unlikely if the hash function maps adjacent pages to adjacent locations in the hash table. The new page buffer collects statistics allowing us to test this.

To optimize scans of all entries in the page buffer, all entries are also stored in the doubly linked index list.

The replacement policy is a modified version of LRU with second pass for dirty entries. It differs from the standard version in that the user is allowed to reserve a percentage of the pages for raw data and/or metadata[18], and when operating in VFD SWMR mode, as required by functional requirements 1, 2, and 3 above.

When operating in VFD SWMR mode, the new buffer cache also maintains two additional lists -- the tick list, and the delayed write list.

Whenever a page or multi-page metadata entry is modified during a tick, it is placed on the tick list. If, in addition, the write of the entry must be delayed for one or more ticks, the entry is also removed from the LRU and inserted on the delayed write list.

At the end of each tick, all entries are removed from the tick list and the metadata file index is updated. Multi-page metadata entries that are not subject to delayed write constraints are flushed and evicted immediately.

Also at the end of each tick, the delayed write list is searched for entries whose write delays have expired. Any such multi-page metadata entries are flushed and evicted. Regular pages whose write delays have expired are simply moved to the LRU where they may be flushed and evicted as normal.

While the new page buffer tracks the total, clean, and dirty page buffer size, at present, it does not track additions since the beginning of the current tick, or provide a mechanism to support triggering the early end of tick.

Further implementation details are discussed in the header comments for main structure of the new page buffer (`H5PB_t`) and for entries in the page buffer (`H5PB_entry_t`). These header comments and the associated definitions are reproduced below.

```
/****************************************************************************
 *
 * structure H5PB_t
 *
 * Catchall structure for all variables specific to an instance of the page
 * buffer.
 *
 * At present, the page buffer serves two purposes in the HDF5 library.
 *
 * Under normal operating conditions, it serves as a normal page buffer whose
 * purpose is to minimize and optimize file I/O by aggregating small metadata
 * and raw data writes into pages, and by caching frequently used pages.
 *
 * In addition, when a file is opened for VFD SWMR writing, the page buffer is
 * used to retain copies of all metadata pages and multi-page metadata entries
 * that are written in a given tick, and under certain cases, to delay metadata
 * page and/or multi-page metadata entry writes for some number of ticks.
 * If the entry has not appeared in the VFD SWMR index for at least max_lag
 * ticks, this is necessary to avoid message from the future bugs.  See the
 * VFD SWMR RFC for further details.
 *
 * To reflect this, the fields of this structure are divided into three
```

[18] This option was introduced in the original version of the page buffer. It is supported in the new page buffer as doing so allowed us to reuse the existing test code – a time saver in the phase 1 implementation. Whether this option is of sufficient value as to justify its retention is an open question to which some thought should be given.

```
 * sections.  Specifically fields needed for general operations, fields needed
 * for VFD SWMR, and statistics.
 *
 * FIELDS FOR GENERAL OPERATIONS:
 *
 * magic:       Unsigned 32 bit integer that must always be set to
 *              H5PB__H5PB_T_MAGIC.  This field is used to validate pointers to
 *              instances of H5PB_t.
 *
 * page_size:   size_t containing the page buffer page size in bytes.
 *
 * max_pages:   64 bit integer containing the nominal maximum number
 *              of pages in the page buffer.  Note that on creation, the page
 *              buffer is empty, and that under certain circumstances (mostly
 *              related to VFD SWMR) this limit can be exceeded by large
 *              amounts.
 *
 * curr_pages:  64 bit integer containing the current number of pages
 *              in the page buffer.  curr_pages must always equal the sum of
 *              curr_md_pages + curr_rd_pages.
 *
 *              Note that in the context of VFD SWMR, this count does NOT
 *              include multi-page metadata entries.
 *
 * curr_md_pages: 64 bit integer containing the current number of
 *              metadata pages in the page buffer.
 *
 *              Note that in the context of VFD SWMR, this count does NOT
 *              include multi-page metadata entries.
 *
 * curr_rd_pages: 64 bit integer containing the current number of
 *              raw data pages in the page buffer.
 *
 * min_md_pages: 64 bit integer containing the number of pages in the
 *              page buffer reserved for metadata.  No metadata page may be
 *              evicted from the page buffer if curr_md_pages is less than or
 *              equal to this value.
 *
 * min_rd_pages: 64 bin integer containing the number of pages in the
 *              page buffer reserved for raw data.  No page or raw data may be
 *              evicted from the page buffer if curr_rd_pages is less than or
 *              equal to this value.
 *
 * The FAPL fields are used to store the page buffer configuration data
 * provided to the page buffer in the H5PB_create() call.
 *
 * max_size:    Maximum page buffer size supplied by the FAPL.
 *
 * min_meta_perc: Percent of the page buffer reserved for metadata as
 *              supplied in the FAPL.
 *
 * min_raw_perc: Percent of the page buffer reserved for metadata as
 *              supplied in the FAPL.
 *
 * The purpose of the index is to allow us to efficiently look up all pages
 * (and multi-page metadata entries in the context of VFD SWMR) in the
 * page buffer.
 *
 * This function is provided by a hash table with chaining, albeit with one
 * un-unusual feature.
 *
 * Specifically hash table size must be a power of two, and the hash function
 * simply clips the high order bits off the page offset of the entry.
 *
 * This should work, as space is typically allocated sequentually, and thus
 * via a reverse principle of locality argument, hot pages are unlikely to
 * hash to the same bucket.  That said, we must collect statistics to alert
 * us should this not be the case.
 *
 * We also maintain a linked list of all entries in the index to facilitate
 * flush operations.
 *
 * index        Array of pointer to H5PB_entry_t of size
 *              H5PB__HASH_TABLE_LEN.  This size must ba a power of 2,
 *              not the usual prime number.
 *
 * index_len:   Number of entries currently in the hash table used to index
 *              the page buffer.  index_len should always equal
 *              clean_index_len + dirty_index_len.
 *
 * clean_index_len: Number of clean entries currently in the hash table
 *              used to index the page buffer.
 *
 * dirty_index_len: Number of dirty entries currently in the hash table
 *              used to index the page buffer.
 *
 * index_size:  Number of bytes currently stored in the hash table used to
 *              index the page buffer.  Under normal circumstances, this
 *              value will be index_len * page size.  However, if
 *              vfd_swmr_writer is TRUE, it may be larger.
 *
 *              index_size should always equal clean_index_size +
 *              dirty_index_size.
 *
 * clean_index_size: Number of bytes of clean entries currently stored in
 *              the hash table used to index the page buffer.
 *
 * dirty_index_size: Number of bytes of dirty entries currently stored in
 *              the hash table used to index the page buffer.
 *
 * il_len:      Number of entries on the index list.
 *
 *              This must always be equal to index_len.  As such, this
 *              field is redundant.  However, the existing linked list
 *              management macros expect to maintain a length field, so
 *              this field exists primarily to avoid adding complexity to
 *              these macros.
 *
 * il_size:     Number of bytes of cache entries currently stored in the
 *              index list.
 *
 *              This must always be equal to index_size.  As such, this
 *              field is redundant.  However, the existing linked list
 *              management macros expect to maintain a size field, so
 *              this field exists primarily to avoid adding complexity to
 *              these macros.
 *
 * il_head:     Pointer to the head of the doubly linked list of entries in
```

```
*              the index list.  Note that cache entries on this list are
*              linked by their il_next and il_prev fields.
*
*              This field is NULL if the index is empty.
*
* il_tail:     Pointer to the tail of the doubly linked list of entries in
*              the index list.  Note that cache entries on this list are
*              linked by their il_next and il_prev fields.
*
*              This field is NULL if the index is empty.
*
*
* Fields supporting the modified LRU policy:
*
* See most any OS text for a discussion of the LRU replacement policy.
*
* Under normal operating circumstances (i.e. vfd_swmr_writer is FALSE)
* all entries will reside both in the index and in the LRU.  Further,
* all entries will be of size page_size.
*
* The VFD SWMR writer case (i.e. vfd_swmr_writer is TRUE) is complicated
* by the requirements that we:
*
* 1) buffer all metadata writes (including multi-page metadata writes) that
*    occur during a tick, and
*
* 2) when necessary, delay metadata writes for up to max_lag ticks to
*    avoid message from the future bugs on the VFD SWMR readers.
*
* See discussion of fields supporting VFD SWMR below for details.
*
* Discussions of the individual fields used by the modified LRU replacement
* policy follow:
*
* LRU_len:     Number of page buffer entries currently on the LRU.
*
*              Observe that LRU_len + dwl_len must always equal
*              index_len.
*
* LRU_size:    Number of bytes of page buffer entries currently residing
*              on the LRU list.
*
*              Observe that LRU_size + dwl_size must always equal
*              index_size.
*
* LRU_head_ptr:  Pointer to the head of the doubly linked LRU list.  Page
*              buffer entries on this list are linked by their next and
*              prev fields.
*
*              This field is NULL if the list is empty.
*
* LRU_tail_ptr:  Pointer to the tail of the doubly linked LRU list.  Page
*              buffer entries on this list are linked by their next and
*              prev fields.
*
*              This field is NULL if the list is empty.
*
*
* FIELDS SUPPORTING VFD SWMR:
*
* If the file is opened as a VFD SWMR writer (i.e. vfd_swmr_writer == TRUE),
* the page buffer must retain the data necessary to update the metadata
* file at the end of each tick, and also delay writes as necessary so as
* to avoid message from the future bugs on the VFD SWMR readers.
*
* The tick list exists to allow us to buffer copies of all metadata writes
* during a tick, and the delayed write list supports delayed writes.
*
* If a regular page is written to during a tick, it is placed on the tick
* list.  If there is no reason to delay its write to file (i.e. either
* it was just allocated, or it has existed in the metadata file index for
* at least max_lag ticks), it is also placed on the LRU, where it may be
* flushed, but not evicted.  If its write must be delayed, it is placed on
* the delayed write list, where it must remain until its write delay is
* satisfied -- at which point it is moved to the LRU.
*
* If a multi-page metadata entry is written during a tick, it is placed on
* the tick list.  If, in addition, the write of the entry must be delayed,
* it is also place on the delayed write list.  Note that multi-page metadata
* entries may never appear on the LRU.
*
* At the end of each tick, the tick list is emptied.
*
* Regular pages are simply removed from the tick list, as they must already
* appear on either the LRU or the delayed write list.
*
* Multi-page metadata entries that are not also on the delayed write list
* are simply flushed and evicted.
*
* The delayed write list is also scanned at the end of each tick.  Regular
* entries that are now flushable are placed at the head of the LRU.  Multi-
* page metadata entries that are flushable are flushed and evicted.
*
* The remainder of this sections contains discussions of the fields and
* data structures used to support the above operations.
*
* vfd_swmr_writer: Boolean flag that is set to TRUE iff the file is
*                  the file is opened in VFD SWMR mode.  The remaining
*                  VFD SWMR fields are defined iff vfd_swmr_writer is TRUE.
*
* mpmde_count: int64_t containing the number of multi-page metadata
*                  entries currently resident in the page buffer.  Observe
*                  that index_len should always equal curr_pages + mpmde_count.
*
* cur_tick:        uint64_t containing the current tick.  This is a copy of
*                  the same field in the associated instance of H5F_file_t,
*                  and is maintained as a convenience.
*
* In the context of VFD SWMR the delayed write list allows us to delay
* metadata writes to the HDF5 file until it appears in all indexes in the
* last max_lag ticks.  This is essential if a version of the page or
* multi-page metadata entry already exists in the HDF5 file -- failure to
* delay the write can result in a message from the future which will
* likely be perciived as file corruption by the reader.
*
* To facilitate identification of entries that must be removed from the
* DWL during the end of tick scan, the list always observes the following
* invarient for any entry on the list:
```

```
 *
 *    entry_ptr->next == NULL ||
 *    entry_ptr->delay_write_until >= entry_ptr->next->delay_write_until
 *
 * Discussion of the fields used to implement the delayed write list follows:
 *
 * max_delay:   Maximum of the delay_write_until fields of the entries on
 *              the delayed write list.  This must never be more than max_lag
 *              ticks in advance of the current tick, and should be set to
 *              zero if the delayed write list is empty.
 *
 * dwl_len:     Number of page buffer entries currently on the delayed
 *              write list.
 *
 *              Observe that LRU_len + dwl_len must always equal
 *              index_len.
 *
 * dwl_size:    Number of bytes of page buffer entries currently residing
 *              on the DWL.
 *
 *              Observe that LRU_size + dwl_size must always equal
 *              index_size.
 *
 * dwl_head_ptr:  Pointer to the head of the doubly linked delayed write list.
 *              Page buffer entries on this list are linked by their next and
 *              prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * dwl_tail_ptr:  Pointer to the tail of the doubly linked delayed write list.
 *              Page buffer entries on this list are linked by their next and
 *              prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * For VFD SWMR to function, copies of all pages modified during a tick must
 * be retained in the page buffer to allow correct updates to the index and
 * metadata file at the end of tick.
 *
 * To implement this, all entries modified during the current tick are placed
 * on the tick list.  Entries are removed from the tick list during end of
 * tick processing, so each tick starts with an empty tick list.
 *
 * Unless the entry also resides on the delayed write list, entries on the
 * tick list may be flushed, but they may not be evicted.
 *
 * Discussion of the fields used to implement the tick list follows:
 *
 * tl_len:      Number of page buffer entries currently on the tick list
 *
 * tl_size:     Number of bytes of page buffer entries currently residing
 *              on the tick list.
 *
 * tl_head_ptr:  Pointer to the head of the doubly linked tick list.
 *              Page buffer entries on this list are linked by their tl_next
 *              and tl_prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * tl_tail_ptr:  Pointer to the tail of the doubly linked tick list.
 *              Page buffer entries on this list are linked by their tl_next
 *              and tl_prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 *
 * STATISTICS:
 *
 * Multi-page metadata entries (which may only appear in VFD
 * SWMR mode) are NOT counted in the following statistics.
 *
 * Note that all statistics fields contain only data since the last time
 * that statistics were reset.
 *
 * bypasses:    Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of times that the page buffer has been
 *              bypassed for raw data, metadata, and for multi-page
 *              metadata entries (VFD SWMR only) as indexed by H5PB__STATS_MD,
 *              H5PB__STATS_RD, and H5PB__STATS_MPMDE respectively.
 *
 * accesses:    Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer accesses for raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by H5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * hits:        Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer hits for raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * misses:      Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer misses for raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * loads:       Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer loads for raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * insertions:  Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer insertions of raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * flushes:     Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer flushes of raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * evictions:   Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer evictions of raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * clears:      Array of int64_t of length H5PB__NUM_STAT_TYPES containing
 *              the number of page buffer entry clears of raw data, metadata,
 *              and for multi-page metadata entries (VFD SWMR only) as
 *              indexed by 5PB__STATS_MD, H5PB__STATS_RD, and
 *              H5PB__STATS_MPMDE respectively.
 *
 * max_lru_len: int64_t containing the maximum number of entries that
 *              have appeared in the LRU.
 *
 * max_lru_size: int64_t containing the maximum size of the LRU.
 *
 * lru_md_skips: When searching for an entry to evict, metadata entries on
 *              the LRU must be skipped if the number of metadata pages
 *              in the page buffer fails to exceed min_md_pages.
 *
 *              This int64_t is used to keep a count of these skips.
 *
 *              If this number becomes excessive, it will be necessary to
 *              add a holding tank for such entries.
 *
 * lru_rd_skips: When searching for an entry to evict, raw data entries on
 *              the LRU must be skipped if the number of raw data pages
 *              in the page buffer fails to exceed min_rd_pages.
 *
 *              This int64_t is used to keep a count of these skips.
 *
 *              If this number becomes excessive, it will be necessary to
 *              add a holding tank for such entries.
 *
 * Multi-page metadata entries (which appear only in VFD SWMR mode) are
 * listed in the hash take, and thus they are counted in the following
 * statistics.
 *
 * total_ht_insertions: Number of times entries have been inserted into the
 *              hash table.
 *
 * total_ht_deletions: Number of times entries have been deleted from the
 *              hash table.
 *
 * successful_ht_searches: int64 containing the total number of successful
 *              searches of the hash table.
 *
 * total_successful_ht_search_depth: int64 containing the total number of
 *              entries other than the targets examined in successful
 *              searches of the hash table.
 *
 * failed_ht_searches: int64 containing the total number of unsuccessful
 *              searches of the hash table.
 *
 * total_failed_ht_search_depth: int64 containing the total number of
 *              entries examined in unsuccessful searches of the hash
 *              table.
 *
 * max_index_len:  Largest value attained by the index_len field.
 *
```

```
 * max_clean_index_len:  Largest value attained by the clean_index_len field.
 *
 * max_dirty_index_len:  Largest value attained by the dirty_index_len field.
 *
 * max_index_size:  Largest value attained by the index_size field.
 *
 * max_clean_index_size:  Largest value attained by the clean_index_size field.
 *
 * max_dirty_index_size:  Largest value attained by the dirty_index_size field.
 *
 * max_rd_pages: Maximum number of raw data pages in the page buffer.
 *
 * max_md_pages: Maximum number of metadata pages in the page buffer.
 *
 *
 * Statistics pretaining to VFD SWMR.
 *
 * max_mpmde_count: Maximum number of multi-page metadata entries in the
 *              page buffer.
 *
 * lru_tl_skips: When searching for an entry to evict, metadata entries on
 *              the LRU must be skipped if they also reside on the tick list.
 *
 *              This int64_t is used to keep a count of these skips.
 *
 *              If this number becomes excessive, it will be necessary to
 *              add a holding tank for such entries.
 *
 * max_tl_len:  int64_t containing the maximum value of tl_len.
 *
 * max_tl_size: int64_t containing the maximum value of tl_size.
 *
 * delayed_writes: int64_t containing the total number of delayed writes.
 *
 * total_delay: int64_t containing the total number of ticks by which
 *              entry writes have been delayed.
 *
 * max_dwl_len: int64_t containing the maximum value of dwl_len.
 *
 * max_dwl_size: int64_t containing the maximum value of dwl_size.
 *
 * total_dwl_ins_depth: int64_t containing the total insertion depth
 *              required to maintain the odering invarient on the
 *              delayed write list.
 *
 *******************************************************************************/

#define H5PB__H5PB_T_MAGIC  0x01020304

#define H5PB__STATS_MD          0
#define H5PB__STATS_RD          1
#define H5PB__STATS_MPMDE       2
#define H5PB__NUM_STAT_TYPES    3

typedef struct H5PB_t {

    /* Fields for general operations: */

    uint32_t magic;
    size_t page_size;
    int64_t max_pages;
    int64_t curr_pages;
    int64_t curr_md_pages;
    int64_t curr_rd_pages;
    int64_t min_md_pages;
    int64_t min_rd_pages;

    /* FAPL fields */
    size_t max_size;
    unsigned min_meta_perc;
    unsigned min_raw_perc;

    /* index */
    H5PB_entry_t *(ht[H5PB__HASH_TABLE_LEN]);
    int64_t index_len;
    int64_t clean_index_len;
    int64_t dirty_index_len;
    int64_t index_size;
    int64_t clean_index_size;
    int64_t dirty_index_size;
    int64_t il_len;
    int64_t il_size;
    H5PB_entry_t * il_head;
    H5PB_entry_t * il_tail;

    /* LRU */
    int64_t LRU_len;
    int64_t LRU_size;
    H5PB_entry_t * LRU_head_ptr;
    H5PB_entry_t * LRU_tail_ptr;


    /* Fields for VFD SWMR operations: */

    hbool_t vfd_swmr_writer;
    int64_t mpmde_count;
    uint64_t cur_tick;

    /* delayed write list */
    uint64_t max_delay;
    int64_t dwl_len;
    int64_t dwl_size;
    H5PB_entry_t * dwl_head_ptr;
    H5PB_entry_t * dwl_tail_ptr;

    /* tick list */
    int64_t tl_len;
    int64_t tl_size;
    H5PB_entry_t * tl_head_ptr;
    H5PB_entry_t * tl_tail_ptr;

    /* Statistics: */

    /* general operations statistics: */
    /* these statistics count pages only, not multi-page metadata entries
     * (that occur only in the VFD SWMR writer case).
     */
    int64_t bypasses[H5PB__NUM_STAT_TYPES];
    int64_t accesses[H5PB__NUM_STAT_TYPES];
    int64_t hits[H5PB__NUM_STAT_TYPES];
    int64_t misses[H5PB__NUM_STAT_TYPES];
    int64_t loads[H5PB__NUM_STAT_TYPES];
    int64_t insertions[H5PB__NUM_STAT_TYPES];
    int64_t flushes[H5PB__NUM_STAT_TYPES];
    int64_t evictions[H5PB__NUM_STAT_TYPES];
    int64_t clears[H5PB__NUM_STAT_TYPES];
    int64_t max_lru_len;
    int64_t max_lru_size;
    int64_t lru_md_skips;
    int64_t lru_rd_skips;

    /* In the VFD SWMR case, both pages and multi-page metadata entries
     * are stored in the index.  Thus mult-page metadata entries are
     * included in the index related statistics.
     */
    int64_t total_ht_insertions;
    int64_t total_ht_deletions;
    int64_t successful_ht_searches;
    int64_t total_successful_ht_search_depth;
    int64_t failed_ht_searches;
    int64_t total_failed_ht_search_depth;
    int64_t max_index_len;
    int64_t max_clean_index_len;
    int64_t max_dirty_index_len;
    int64_t max_index_size;
    int64_t max_clean_index_size;
    int64_t max_dirty_index_size;
    int64_t max_rd_pages;
    int64_t max_md_pages;


    /* vfd swmr statistics */
    int64_t max_mpmde_count;
    int64_t lru_tl_skips;
    int64_t max_tl_len;
    int64_t max_tl_size;
    int64_t delayed_writes;
    int64_t total_delay;
    int64_t max_dwl_len;
    int64_t max_dwl_size;
    int64_t total_dwl_ins_depth;

} H5PB_t;

/****************************************************************************
 *
 * structure H5PB_entry_t
 *
 * Individual instances of the H5PB_entry_t structure are used to manage
 * individual pages in the page buffer.  In the case of a VFD SWMR writer,
 * they are also used to manage multi-page metadata entries.
 *
 * The fields of this structure are discussed below:
 *
 *                                              JRM - 9/27/18
 *
 * magic:       Unsigned 32 bit integer that must always be set to
 *              H5PB__H5PB_ENTRY_T_MAGIC when the entry is valid.
 *
 * pb_ptr:      Pointer to the page buffer that contains this entry.
 *
 * addr:        Base address of the page in the file.
 *
 * page:        Page offset of the page -- i.e. addr / pb_ptr->page_size.
 *              Note that addr must always equal page * pb_ptr->page_size.
 *
 * size:        Size of the page buffer entry in bytes.  Under normal
 *              circumstance, this will always be equal to pb_ptr->page_size.
 *              However, in the context of a VFD SWMR writer, the page
 *              buffer may be used to store multi-page metadata entries
 *              until the end of tick, or to delay writes of such entries
 *              for up to max_lag ticks.
 *
 *              In such cases, size must be greater than pb_ptr->page_size.
 *
 * image_ptr:   Pointer to void.  When not NULL, this field points to a
 *              dynamically allocated block of size bytes in which the
 *              on disk image of the page.  In the context of VFD SWMR,
 *              it points to the image of the multi-page metadata entry.
 *
 * mem_type:    Type (H5F_mem_t) of the page buffer entry.  This value
 *              is needed when reading or writing the entry from/to file.
 *
 * is_metadata:  Boolean flag that is set to TRUE iff the associated
 *              entry is a page of metadata (or, in the context of VFD
 *              SWMR, a multi-page metadata entry).
 *
 * is_dirty:    Boolean flag indicating whether the contents of the page
 *              buffer entry has been modified since the last time it
 *              was written to disk.
 *
 *
 * Fields supporting the hash table:
 *
 * Entries in the page buffer are indexed by a more or less conventional
 * hash table with chaining (see header comment on H5PB_t for futher details).
 * If there are multiple entries in any hash bin, they are stored in a doubly
 * linked list.
 *
 * To facilitate flushing the page buffer, we also maintain a doubly linked
 * list of all entries in the page buffer.
 *
 * ht_next:     Next pointer used by the hash table to store multiple
 *              entries in a single hash bin.  This field points to the
 *              next entry in the doubly linked list of entries in the
 *              hash bin, or NULL if there is no next entry.
 *
 * ht_prev:     Prev pointer used by the hash table to store multiple
 *              entries in a single hash bin.  This field points to the
 *              previous entry in the doubly linked list of entries in
 *              the hash bin, or NULL if there is no previuos entry.
 *
 * il_next:     Next pointer used by the index to maintain a doubly linked
 *              list of all entries in the index (and thus in the page buffer).
 *              This field contains a pointer to the next entry in the
 *              index list, or NULL if there is no next entry.
```

```
*
* il_prev:     Prev pointer used by the index to maintain a doubly linked
*              list of all entries in the index (and thus in the page buffer).
*              This field contains a pointer to the previous entry in the
*              index list, or NULL if there is no previous entry.
*
*
* Fields supporting replacement policies:
*
* The page buffer must have a replacement policy, and it will usually be
* necessary for this structure to contain fields supporting that policy.
*
* At present, only a modified LRU replacement policy is contemplated,
* (see header comment for H5PB_t for details), for which the following
* fields are adequate.
*
* next:        Next pointer in either the LRU, or (in the context of
*              VFD SWMR) the delayed write list.  If there is no next entry
*              on the list, this field should be set to NULL.
*
* prev:        Prev pointer in either the LRU, or (in the context of
*              VFD SWMR) the delayed write list.  If there is no previous
*              entry on the list, this field should be set to NULL.
*
* Fields supporting VFD SWMR:
*
* is_mpmde:    Boolean flag that is set to TRUE iff the entry
*              is a multi-page metadata entry.  In the absense of VFD
*              SWMR, the field should always be set to FALSE.
*
*              Observe that:
*
*              is_mpmde <==> is_metadata && size > pb_ptr->page_size
*
* loaded:      Boolean flag that is set to TRUE iff the entry was loaded
*              from file.  This is a necessary input in determining
*              whether the write of the entry must be delayed.
*
*              This field is only maintained in the VFD SWMR case
*              and should be false otherwise.
*
* modified_this_tick:  This field is set to TRUE iff pb_ptr->vfd_swrm_write
*              and the entry has been modified in the current tick.  If
*              modified_this_tick is TRUE, the entry must also be in the
*              tick list.
*
* delay_write_until: Unsigned 64 bit integer containing the first tick
*              in which the entry may be written to file, or 0 if there
*              is no such constraint.  It should be set ot 0 when VFD
*              is not enabled.
*
* tl_next:     Next pointer on the list of entries modified in the current
*              tick,  If the enty is not on the tick list, or if there is
*              no next entry on the list, this field should be set to NULL.
*
* tl_prev:     Prev pointer on the list of entries modified in the current
*              tick,  If the enty is not on the tick list, or if there is
*              no previous entry on the list, this field should be set to
*              NULL.
```

```
 *
 ****************************************************************************/

#define H5PB__H5PB_ENTRY_T_MAGIC  0x02030405

struct H5PB_entry_t {

    uint32_t                    magic;
    H5PB_t                     *pb_ptr;
    haddr_t                     addr;
    uint64_t                    page;
    size_t                      size;
    void                       *image_ptr;
    H5FD_mem_t                  mem_type;
    hbool_t                     is_metadata;
    hbool_t                     is_dirty;

    /* fields supporting the hash table: */
    struct H5PB_entry_t        *ht_next;
    struct H5PB_entry_t        *ht_prev;
    struct H5PB_entry_t        *il_next;
    struct H5PB_entry_t        *il_prev;

    /* fields supporting replacement policies: */
    struct H5PB_entry_t        *next;
    struct H5PB_entry_t        *prev;

    /* fields supporting VFD SWMR */
    hbool_t                     is_mpmde;
    hbool_t                     loaded;
    hbool_t                     modified_this_tick;
    uint64_t                    delay_write_until;
    struct H5PB_entry_t        *tl_next;
    struct H5PB_entry_t        *tl_prev;

}; /* H5PB_entry_t */
```

## 3.5 Metadata File Management

### 3.5.1 Metadata File Format

The metadata file format is constructed so as to allow the VFD SWMR reader VFD to intercept metadata page reads and satisfy them with a consistent (but possibly dated) view of the HDF5 file metadata. Further, this view of the metadata must remain consistent even if the reader falls up to max_lag ticks behind the VFD SWMR writer.

Thus, at the metadata file level, we must:

- Ensure that no metadata page or multipage piece of metadata in the metadata file is overwritten until it has not appeared in the current index for at least max_lag ticks.
- Ensure that all metadata pages and/or multipage pieces of metadata dirtied in the current tick are written to the metadata file before the index for the current tick becomes visible.

As shall be seen, we will use POSIX file I/O semantics (combined with checksums and retries when necessary) to guarantee this in the POSIX case, and atomic writes of metadata file change lists in the NFS and object store cases.

However, before discussing the exact particulars of writing and reading the metadata file, we must first define its format and free space management.

### *3.5.1.1 Metadata File Header*

The Metadata File Header must be located at offset 0 in the metadata file, and has the following format:

Metadata File Header:

| byte | byte | byte | byte |
|---|---|---|---|
| Signature | | | |
| Page Size | | | |
| Tick Number | | | |
| Index Offset | | | |
| Index Length | | | |
| checksum | | | |

The fields of the metadata file header are described in the following table.

| Field Name: | Description: |
|---|---|
| Signature | Magic number indicating that this is a VFD SWMR metadata file header. Must be set to 'VHDR'. |
| Page Size | Size of pages in both the HDF5 file and the Metadata file in bytes. |
| Tick Number | Sequence number of the current tick. This is an unsigned 64 bit value that is initialized to zero on file creation / open, and incremented by the VFD SWMR writer at the end of each tick. |
| Index Offset | Unsigned 64-bit value containing the offset of the current metadata file index in the metadata file in bytes.<br>Ideally, the index will be located immediately after the header – in which case this value will be the offset of the first byte after the header.<br>However, regardless of how much space is reserved for the header and index, it is always possible that the index will become too large for it. In this case, this field contains the page aligned base address of the index. Note that the index must reside in a contiguous sequence of pages. |

| | |
|---|---|
| Index Length | Unsigned 64-bit value containing the length of the current metadata file index in bytes. |
| checksum | Checksum of the contents of the Metadata File header. |

Ideally the index offset and length fields would be of the sizes specified in the superblock of the HDF5 file for offsets and lengths. However, this data may not be available to the reader when the metadata file is first read. Thus both of these values are 8 bytes – the maximum value with current file systems.

Similarly, the page size stored in the HDF5 file may not be accessible to the reader when the reader VFD first accesses the metadata file, and thus must be listed in the header.

#### *3.5.1.2 Metadata File Index*

The Metadata File Index file format is variable length, with its length being determined by the number of entries in the index. The top level of the format is shown first, with the format of individual index entries given subsequently.

Metadata File Index:

| byte | byte | byte | byte |
|---|---|---|---|
| Signature | | | |
| Tick Number | | | |
| Number of Entries | | | |
| Index Entry 0 | | | |
| . | | | |
| . | | | |
| . | | | |
| Index Entry n | | | |
| checksum | | | |

The fields of the top level format are described in the following table. Recall that the "Index Entry" fields are a sub-formats embedded in the Metadata File Index format.

| Field Name: | Description: |
|---|---|
| Signature | Magic number indicating that this is a VFD SWMR metadata file index. Must be set to 'VIDX'. |
| Tick Number | Sequence number of the current tick. This is an unsigned 64 bit value that |

| | |
|---|---|
| | is initialized to zero on file creation / open, and incremented by the VFD SWMR writer at the end of each tick. |
| Number of Entries | Unsigned 32 bit value containing the number of entries in the index. Note that if an existing file is opened for VFD SWMR write, this value will be zero until such time as metadata is modified by the VFD SWMR writer. |
| Index Entry n | N'th entry in the index.<br>See "Metadata File Index Entry" below for the details of these fields.<br>Index entries must be sorted in increasing HDF5 file page offset. |
| checksum | Checksum of the contents of the Metadata File Index. |

The Metadata File Index Entry is a fixed length format. Its structure is described below:

Metadata File Index Entry:

| byte | byte | byte | byte |
|---|---|---|---|
| HDF5 File Page Offset | | | |
| Metadata File Page Offset | | | |
| Length | | | |
| Entry Checksum | | | |

| Field Name: | Description: |
|---|---|
| HDF5 File Page Offset | Unsigned 32-bit value containing the base address of the metadata page, or multi page metadata entry in the HDF5 file IN PAGES. To obtain byte offset, multiply this value by Page Size in the Metadata File Header. |
| Metadata File Page Offset | Unsigned 32-bit value containing the base address of the metadata page, or multi page metadata entry in the metadata file IN PAGES. To obtain byte offset, multiply this value by Page Size in the Metadata File Header. |
| Length | Unsigned 32-bit value containing the length of the metadata page or multi page metadata entry IN BYTES. If this is a metadata page, the Length must equal the page size. If this is an individual multi page cache entry, the length must be greater than the page size, but need not be a multiple of the page size |
| Entry Checksum | Unsigned 32-bit value containing the checksum of the referenced metadata page or multi-page metadata entry. |

Observe that the offsets are listed in pages, not bytes, and that 32 bit fields are used for these values. Assuming a 4 KB page, this means that the maximum HDF5 file size supported by this metadata file index format is 16 TB (2 TB with a 512 byte page size). While this should be sufficient for now, there will be use cases in which it is insufficient.

Fortunately, the metadata file is discarded on HDF5 file close, so there are no forward / backward compatibility issues.[19]

Minimizing the size of the index is important for performance, so we will likely address this issue by choosing the metadata file index format based on page size and a user supplied hint on maximum file size. TODO: Work out the details of this. Also consider how we might avoid writing the entire index by publishing deltas instead of the entire index.

Length is also an unsigned 32-bit value, which limits the maximum size of mulit-page pieces of metadata to 4 GB. Since the largest piece of metadata seen in the wild was ~100 MB, this limitation shouldn't bite us for quite a while.

#### *3.5.1.3 Metadata File Body*

The metadata file body is simply a page-aligned list of metadata pages and multi-page metadata entries. The current snapshot of the HDF5 file metadata is given by that subset of these metadata pages and multi-page entries listed in the current index. Metadata pages and multi-page metadata entries that are not listed in the index must be retained in the metadata file until they have not appeared in the index for at least max_lag ticks. This ensures that indexes will be valid for at least max_lag ticks.

#### *3.5.1.4 Metadata File Free Space Management*

To copy a metadata page or multi-page metadata entry into the metadata file, we must first allocate space for it. Similarly, to control the size of the metadata file, we must eventually reuse metadata file space allocated to obsolete pages or multi-page metadata entries. The metadata file free space manager must support these operations by allocating space and accepting freed space for re-use.

If a metadata page or mult-page metadata entry is modified, it must be retained in the metadata file for at least max_lag ticks, so as to allow for readers that are up to max_lag ticks behind the writer. To support this, the offset and length of superseded metadata pages or multi-page metadata entries must be placed at the head of a doubly linked list, decorated with the number of the tick in which they were superseded. Call this list the delayed free space release linked list.

End of tick processing for the VFD SWMR for the writer must scan the delayed free space release linked list from the bottom up, release to the metadata file free space manager all the space that has resided on the linked list for more than max_lag ticks, and remove the associated entries from the list.

##### *3.5.1.4.1 Design for Metadata File Free Space Manager*

The metadata file free space manager must satisfy the following functional requirements:

[19] True if the reader and writer use the same HDF5 release. If we choose to allow the case where this is not TRUE, we probably need to add version and page offset width fields to the header.

- Allocate the requested number of contiguous pages in the metadata file, extending the file if necessary.
- Accept blocks of one or more released pages and add them to the free list. Free space should be coalesced where possible.
- At least for the first allocation, space must be allocated at the head of the file. Thus the first allocation will be for one or more pages at offset zero for the metadata file header and index.

The HDF5 library has already implemented the free-space module (H5FS) for handling free-space, and the existing clients are the free-space managers for file space (H5MF) and the fractal heap (H5HF).

Similarly, we will create free-space manager for handling free-space in the metadata file (H5MV) as a client of the H5FS module. As we will throw away the metadata file when the writer closes for VFD SWMR, the free-space manager does not need to be persistent.

###### *3.5.1.4.1.1 Initialization*

- Add a field to the *H5F_file_t* structure, which will point to the free-space manager for the metadata file:
  - H5FS_t *mv_fspace
- Initialize the field to NULL in *H5F_new()* in *H5Fint.c* on file creation/open

###### *3.5.1.4.1.2 The Free-space Manager Interface*

###### *3.5.1.4.1.2.1 H5MV_alloc()*

Purpose: request space from the metadata file

- If the free-space manager is initialized, search for the requested space from the manager via *H5FS_sect_find()*
  - If a free section is found:
    - Return the address
    - If the section is the same size as the requested space, free the section structure via *H5MV__sect_free()*
    - If the section is larger than the requested space, add the remaining space back to the free-space manager via *H5FS_sect_add()*
- If the free-space manager is not initialized or no free section is found from the manager:
  - For a POSIX file, allocate space by extending the file and then set the new EOA

###### *3.5.1.4.1.2.2 H5MV_xfree()*

Purpose: return free space to the metadata file

- If the free-space manager is not initialized, check if the released space will allow us to shrink the meta-data file via *H5MV_try_shrink()*
- If the space cannot shrink the file, do the following:

  - Create the free-space manager via *H5MV_create()*
  - Add the space to the free-space manager via *H5FS_sect_add()*

###### *3.5.1.4.1.2.3 H5MV_create()*

Purpose: create the free-space manager for the metadata file

- Allocate and initialize the free-space structure via *H5FS_create()*
- The free-space manager will be accessed via *f->shared->mv_fspace*

###### *3.5.1.4.1.2.4 H5MV_try_shrink*

Purpose: check if the space to be freed will shrink the size of the metadata file

- This will be done via the *can_shrink* and the *shrink* section callbacks

###### *3.5.1.4.1.2.5 H5MV_try_extend*

Purpose: check if an allocated block can be extended by a requested size

- If the block adjoins the EOA, extend the file by the requested size and set the new EOA
- If the block adjoins an existing free-space section which fulfills the size requested, extend the block via *H5FS_sect_try_extend()*

###### *3.5.1.4.1.2.6 H5MV_close()*

Purpose: close the free-space manager for the metadata file

- Free the free-space structure via *H5FS_close()*, which will just destroy the section info via *H5FS_sinfo_dest()*

###### *3.5.1.4.1.3 Free-space Section Callbacks*

The section callbacks for the metadata file are set up as follows:

- Define the section class as:
  - H5FS_section_class_t H5MV_FSPACE_SECT_CLS_SIMPLE[1]
- Define callbacks as described below for the *can_merge, merge, can_shrink, shrink,* and *free* class actions
- Set up the routine *H5MV_sect_new()* to create a free-space section structure via *H5FL_MALLOC()* and initialize the section info

###### *3.5.1.4.1.3.1 H5MV__sect_can_merge*

- Check if the two free-space sections adjoin each other
- Return TRUE or FALSE

###### *3.5.1.4.1.3.2 H5MV__sect_merge*

- If the *can_merge* callback returns TRUE, this routine will add the second section's size to the size of the first section, and will free the second section's structure via *H5MV__sect_free()*

###### *3.5.1.4.1.3.3 H5MV_sect_can_shrink*

- Check if the section to be freed is at EOF
- Return TRUE or FALSE

###### *3.5.1.4.1.3.4 H5MV_sect_shrink*

- If the *can_shrink* callback returns TRUE, reduce the file size and set the new EOA

###### *3.5.1.4.1.3.5 H5MV__sect_free*

- Free the section structure via *H5FL_free()*

### 3.5.2 Writing the Metadata File

When creating the metadata file, we will allocate the first `md_pages_reserved` pages of the file to the header and index, where `md_pages_reserved` is >= 1. As long as the header and index fit within this allocation, we can write the header and index in a single atomic write. However, there is always the possibility that header and index will grow to the point that it doesn't fit into any fixed number of pre-allocated pages at the head of the metadata file.

In the initial implementation, we handled this problem by simply aborting if the index size grew too large. While this was adequate for the proof of concept, it is not an acceptable solution for a production version.

For the initial production version, three solutions come to mind:

1. Flush the HDF5 file and replace the index with an empty index.

   If the file was created in VFD SWMR write mode, and has not been flushed previously, this can be done without penalty, as all metadata must be in the metadata file, and listed in the index. Thus there are no concerns for message from the future bus.

   However, if and existing file was opened in VFD SWMR writer mode, or if a file that was created in VFD SWMR writer mode and has already been flushed, there is the possibility that not all metadata is in the metadata file and referenced by the index. Such pieces of metadata are accessed via reads from the HDF5 file proper. If we flush the HDF5 file and replace the index with an empty index, we will create the possibility that we will overwrite older versions of metadata being referenced by any lagging readers – thus creating message from the future bugs.

   We can solve this problem by delaying the flush until all pieces of metadata in the metadata file index have resided in the index for at least `max_lag` ticks. This avoids the possibility of message from the future bugs[20], but compromises any real time guarantees.

2. Allocate space for the index elsewhere in the file. Note that this implies that we can't overwrite the index in place as the header and index can no longer be written in a single atomic action. Instead, we must allocate space for a new index, write it to the metadata file, and then update the header on each tick. Observe that the old index must not be overwritten

[20] i.e. the possibility of reading metadata that was written in a subsequent tick.

for some period of time to allow for the case in which the reader reads the header just before it is overwritten. A delay of `max_lag` ticks is almost certainly excessive, but it simplifies free space management in the metadata file, and thus should be chosen unless we can think of a strong reason to the contrary.

3. Track writes of metadata pages and multi-page metadata entries to the HDF5 file. When this happens, retain the page or multi-page entry in the metadata file index for `max_lag` ticks, and then delete it from the index if there have been no further changes[21].

While there are arguments for all of these options, the first option has the potential to impose irregular delays in end of tick processing – which is inconvenient from a real time perspective. In contrast, 2 and 3 should be manageable in a well constrained amount of time at the end of each tick.

Thus, for the initial production version, we will implement a combination of 2 and 3. Note that we will retain the reservation of space for the index, and not apply 2 unless this reservation becomes inadequate. 3) can be integrated into the scan of the internal representation of the index on writer size, and thus can be low cost.

This point addressed, recall that the metadata file must be written in such a fashion that:

1. All entries in the index are in the metadata file before the index becomes accessible.
2. No entry is overwritten until it has not been mentioned in the current index for at least max_lag ticks.

We have already dealt with requirement 2 with the delayed free space release linked list and free space manager discussed above. This leaves only the first requirement to be addressed in this section. As the solution differs depending on whether we are dealing with a POSIX file system, NFS, or an object store, we address each case in subsections below.

#### *3.5.2.1 POSIX Case*

In a nutshell, writing the metadata file in the POSIX case uses the atomic write and write ordering guarantees of POSIX file I/O semantics to satisfy requirement 1. Note that due to past experience, the VFD SWMR metadata file uses checksums to allow detection of torn writes, and tagging of the header and index with the current tick for sanity checking.

This in turn resolves to the following protocol for updating the metadata file:

1. Allocate space in the metadata file for all metadata pages or multi page metadata entries modified or created in the current tick, and then write the pages or entries to their allocated locations. If the page or entry is an updated version of a page or entry currently listed in the index, insert the old metadata file base address and length at the head of the delayed free space release linked list tagged with the current tick.

   Note: Consider using POSIX vector I/O to minimize the number of function calls.

[21] Again, to use the terminology of `struct H5FD_vfd_swmr_idx_entry_t` presented above, we can remove from the index any metadata page or multi-page metadata cache entry whose `clean` field is `TRUE`, and whose `tick_of_last_flush` is more than `max_lag` ticks in the past.

2. If the header and index fit within the pages reserved for them, overwrite the existing header and index in the metadata file with the current version. Otherwise:
    a. Allocate space for the index and insert the metadata file base address and length of the old index at the head of the delayed free space release linked list tagged with the current tick.
    b. Write the index in its newly allocated location.
    c. Overwrite the existing header.
3. Starting at the bottom of the delayed free space release linked list, scan upwards and release all listed space that is tagged with an index less than or equal to the current index minus max_lag.

While the construction of the updated index and the list of new / modified metadata pages or multi-page entries should be reasonably quick, the file I/O required to update the metadata file could be significant if the tick size is small, and the updates to the metadata file are large. To address this, it may be useful to spawn a thread to handle the metadata file update. We will not do this in the initial production version, but we should write the code that implements the metadata file update with this in mind.

To facilitate passing the metadata file creation / update off to a separate thread, the metadata file update should be handled by a call to

```
herr_t
H5F_update_vfd_swmr_metadata_file(H5F_file_t f,
                                  uint32_t index_len_ptr,
                                  struct H5FD_vfd_swmr_idx_entry_t index[]);
```

which (in the POSIX case) will proceed as follows:

1. Sort the index by increasing offset in the HDF5 file
2. Scan through the sorted index, visiting each entry once, and taking the following actions:
    a. If the entry in the index has a non-NULL `entry_ptr` field:
        i. If it exists, insert the location and length of the previous image of the entry on the delayed free space release linked list
        ii. Allocate space for the entry in the metadata file and update the index
        iii. Compute the checksum of the entry and update the index
        iv. Write the entry into the metadata file
        v. Set the `entry_ptr` field to NULL
    b. If the entry's `moved_to_hdf5_file` field is FALSE, and the entry is clean, and its `tick_of_last_flush` is more than `max_lag` ticks in the past, set the `moved_to_hdf5_file` field to TRUE.

c. If the entry's `moved_to_hdf5_file` field is TRUE, and either the entry is dirty, or its `tick_of_last_flush` is less than or equal to `curr_tick - max_lag`, set the `moved_to_hdf5_file` field to FALSE.

d. If the entry's `moved_to_hdf5_file` field is TRUE, and the entry is clean, and its `tick_of_last_flush` is more than `max_lag` ticks in the past, delete the entry from the index. Do this by reducing the size of the index, and shifting subsequent entries down accordingly as the rest of the index is scanned.

3. Construct the on disk image of the index
4. Write the image of the index to the metadata file
5. Update the header, construct its on disk image, and write the image to the metadata file
6. Release timed out space from the delayed free space release linked list to the free space manager

Modifications for the NFS and object store cases are discussed below.

#### *3.5.2.2 NFS Case*

TBD

#### *3.5.2.3 Object Store Case*

TBD

Note that the object store case will likely be complicated by caching in the object store VFD. If this is the case, the metadata file will have to be supplemented by a similar file indicating allocations, de-allocations and modifications of cached objects in the object store VFD.

### 3.5.3 Reading the Metadata File

Reading the metadata file resolves into two basic operations:

1. Reading the (possibly updated) index
2. Reading a page of metadata or multi-page metadata entry listed in the index

The former operation is directed mostly at determining if the index has been updated, and obtaining the latest version if it has.

The second operation is simply correctly reading the desired version of the metadata page or multi-page entry.

#### *3.5.3.1 POSIX Case*

If we could count on POSIX guarantees in all cases, the POSIX case would be much simpler. However, experience with the current SWMR implementation suggests that we should expect and be able to recover from torn writes (i.e. writes that are supposed to be atomic, but aren't). We are not aware of any difficulties with out of order writes, but prudence suggests that we should design our protocols to detect and manage these as well.

#### *3.5.3.1.1 Reading the Index*

In the initial implementation, we required that the header and index fit within the first `md_pages_reserved` pages of the metadata file, and aborted if this was not the case. This simplified the protocol for obtaining the index and allowed us to minimize the number of reads required to obtain the current version of the index.

While aborting if the index grows too large is not acceptable for a production version, reading the header and index in a single read has performance benefits, and thus we retain the ability of allocating sufficient pages at the head of the metadata file for the header and all expected indexes. However, we must also allow for the possibility that the reserved space will be too small, forcing a relocation of the index.

This complicates reading the index, as may be seen in the following protocol for performing this action.

Define the boolean flag `header_and_index_adjacent` and initialize it to TRUE. Proceed as follows:

1. If `header_and_index_adjacent` is FALSE, goto 7.
2. Load the first `md_pages_reserved` pages of the metadata file into a buffer
3. The metadata file header must be at the start of this buffer. Read it from the buffer and verify its checksum. If the checksum fails, log the error in the log file if it exists, and return to step 2.
4. If the metadata file header indicates that the index starts in the first byte after the header, verify that the header and index together fit in the first `md_pages_reserved` pages of the metadata file. If it does, proceed to step 5. If not, flag an error and abort as this condition should not be possible.

   If the metadata file indicates that the header and index are not adjacent, log this event if the log file is defined, set `header_and_index_adjacent` to FALSE, and goto step 7.
5. Read the index from the buffer and verify its checksum. If the checksum fails, log the error in the log file if it exists, and return to step 2.
6. Verify that the tick number in the header and index match. If they do, goto step 12. If the tick number in the header is one greater than that in the index, we have a very improbable torn write – log it and return to step 2. All other tick number mis-matches should be un-attainable – flag the appropriate error and abort.
7. Load the first header size[22] bytes in the metadata file into a buffer. Note that the remainder of the first page is un-used – although it will probably be filled with junk from the last index to fit in the first `md_pages_reserved` pages of the metadata file.
8. Read the header from the buffer and verify its checksum. If the checksum fails, log the error in the log file if it exists, and return to step 7.
9. Obtain the offset and length of the index from the header. Note that the base address of the index must be page aligned. Load the index into a buffer.

[22] 36 bytes at present.

10. Read the index from the buffer and verify its checksum. If the checksum fails, log the error in the log file if it exists, and abort as this condition indicates either out of order writes by the POSIX file system, a bug in VFD SWMR, or that `max_lag` has been exceeded during this operation.
11. Verify that the tick number in the header and index match. Abort if they do not, as this will indicate either out of order writes by the POSIX file system, a bug in VFD SWMR, or that `max_lag` has been exceeded during this operation.
12. Return the current tick number and index to the caller.

Note that this protocol does not address the possibility of failing if a maximum number of retries is exceeded. We probably want to do this, but this is probably an issue best addressed after we have some operational experience with VFD SWMR. Hence it is deferred for now.

##### *3.5.3.1.2 Reading a Metadata Page or Multi-Page Metadata Entry*

Given the write ordering between new versions of metadata pages or multi-page metadata entries, and the index in which they first appear, torn writes should be impossible, and the only possible failure modes (aside from file system failure) should be:

- Writes to the metadata file completed out of order in violation of POSIX file I/O semantics.
- Read attempted more than max_lag ticks after the last reference to the indicated piece of metadata in the metadata file index.
- Pre-mature reallocation of metadata file space – i.e. an error in the metadata file free space management code.

In all cases, we will treat this as an un-recoverable error. If write order failures prove to be a serious issue, we will have to fall back on some variation on the NFS approach.

Given the offset, length, expected checksum, and a suitably sized buffer, load the specified page(s) from the metadata file into the buffer, and compute the checksum. If the checksum matches the expected checksum, return success. If it doesn't, return failure.

#### ***3.5.3.2 NFS Case***

TBD

#### ***3.5.3.3 Object Store Case***

TBD

## 3.6 Metadata Cache Modifications for Reader

When a VFD SWMR reader detects the start of a new tick and the associated updated index, it must compare the old index with the new, and note any new, or modified entries[23]. For each such new, or

[23] Deleted entries are not an issue, as entries are only removed from the index if they are clean, were last flushed to the HDF5 file more than `max_lag` ticks ago, and are marked as being moved to the HDF5 file in the index prior to their removal. Thus a read of the deleted metadata page or multipage

modified page or multi-page metadata entry, any associated entries in the metadata cache must be invalidated, as they may have been modified.

As mentioned in the conceptual overview, it would be nice if we could simply evict the relevant entries. However, some metadata cache clients are particular about the order in which entries are evicted. Some of this is due to the nature of the cache client in question, and some is an artifact of the existing SWMR implementation. However, unless and until we commit to VFD SWMR and remove the existing SWMR implementation from the library, we have to work with the existing library.

### 3.6.1 Identifying Possibly Modified Metadata Cache Entries

The first step in evicting possibly modified entries from the metadata cache is mapping new / modified pages to lists of entries in the metadata cache.

One obvious solution is to maintain a sorted list of all entries in the metadata cache, and then search for entries whose base addresses fall within the range

[base address of page, (base address of page + page size)].

This is should be do-able via the current skip list facility. However, maintaining and searching this list will impose significant overhead, as the skip list is not exactly a lightweight data structure.

Another option is to construct and maintain a second hash table with a hash function chosen such that all entries in a given page will map to the same bucket. Call this hash table the

page_entry_hash_table.

The hash function would be:

(base_addr_of_MDC_entry / page_size) % hash_table_size

Assuming that both the page size and the hash table sizes are powers of two, this can be computed very efficiently. Use of a free list for hash table entries should minimize malloc / free overhead.

Finally, maintenance of this hash table can be inserted in the existing metadata cache hash table maintenance macros, which should make it very lightweight and easy to implement.

Note, however, that when pages collide, entries from two or more pages will reside in the same bucket. Thus, when scanning the contents of a hash bucket, each entry must be checked to verify that it resides in the target page.

Given the advantages of the page_entry_hash_table approach, this solution was chosen for the initial proof of concept implementation, and will be retained in the first production implementation. If for whatever reason it proves impractical, a skip list of all entries in the metadata cache will be the fallback approach.

Note that there is no need for any special provision for multi-page entries – if such an entry is in the metadata cache, a simple index lookup on its base address will reveal it.

metadata entry from the HDF5 file must return the exact same value at the same read from the prior version of the metadata file. For this reason, entry deletion can be ignored.

### 3.6.2 Evicting Entries that May Have Changed

Before evicting possibly changed entries in the metadata cache, we must first evict all entries in the page buffer that are referenced by new / modified index entries to avoid the possibility of messages from the past. Do this before touching the metadata cache, as our operations on it may trigger reads from the page buffer.

Once we have constructed the list of metadata cache entries that may have changed, we must evict them from the metadata cache.

If the entry in question is not pinned, this is trivial – we just evict it.

However, if the entry is pinned, the client requires that entries be evicted from the metadata cache in some specified order. As mentioned earlier, some of this is an artifact of the existing SWMR implementation, and some is simply due to the structure of the client.

Several ways of dealing with this issue present themselves:

1. Add a refresh() callback to the list of metadata cache client callbacks which would force the client to reload the target entry and adjust any internal structures accordingly.
2. Determine which entries must be evicted before the target entry, evict them, and then evict the target.
3. Via the tagging mechanism, determine what on disk data structure the target entry is part of, and then evict the entire structure.

While addition of the refresh() callback does some violence to the objective of making SWMR completely transparent to the bulk of the HDF5 library, it has the advantage of being very light weight. If we adopt VFD SWMR as our SWMR implementation, we probably want to go this route. However, due to the cost, there is little point in doing so until then.

Option 2) is doable for clients that use the existing flush dependency mechanism to express their flush and eviction ordering requirements to the metadata cache. While I'm not sure, this may be all of them at present. If so, this option should be viable. However, part of the objective of VFD SWMR is to remove the flush dependency facility – thus if we use it, it will be a temporary lash up.

In principle, the mechanism for evicting entire on disk data structures exists via the tagging mechanism, and is tested as part of the EOC (Evict On Close) feature. However, if memory serves, EOC is only implemented for groups and datasets – which suggests that we may have more work to do for the general case.

This said, option 3 was clearly the easiest to implement, and thus was chosen for the initial implementation – with the addition of a refresh function for the superblock, which obviously can't be evicted. While it is heavy weight, we will retain it for the initial production implementation, with the addition of further refresh() functions should they prove necessary. This seems prudent, as we will need to move to the refresh() approach if we commit to VFD SWMR and remove the existing solution. Thus this approach minimizes wasted effort.

### 3.6.3 Possible Optimizations

One obvious optimization is to test possibly modified entries to see if they have actually been modified, and not evict or refresh them if they haven't.

We could do this by decorating metadata cache entries with the checksum of the on disk image of the entry from which the entry was loaded. Since each metadata cache entry knows its offset and length on disk, we could compute the checksum of the entry in the modified metadata page, and only evict (or eventually refresh) the entry if the checksums don't match.

This implies loading the page from the metadata file – but given that the metadata cache contains one or more entries from that page, the chances are that we will need it anyway.

Another possible option is to mark entries as possibly invalidated, and only refresh them if they are accessed. This has the advantage of minimizing reader end of tick processing, and delaying metadata cache entry refresh until the entry is needed.

These notions are listed here so they are not forgotten. However, there are no plans to implement them in the first production implementation.

### 3.7 VFD SWMR Reader VFD

The purpose of the reader VFD in VFD SWMR is to intercept metadata page and multi-page metadata entry reads that appear in the metadata file index, and satisfy them from the metadata file. Metadata reads that don't appear in the metadata file index and all raw data read requests are satisfied from the underlying HDF5 file[24].

Since the reader VFD must open and access the metadata file, and pass un-satisfied read requests to an underlying VFD, the following additional functionality is required:

1. On open, it must:
    a) Wait until the metadata file exists and contains a valid header and index.
    b) Load the initial header and index from the metadata file. Note that in the case of an existing HDF5 file, the tick 0 index will be empty – but there is no requirement that the reader open the file as a VFD SWMR reader at tick 0.
    c) Make the contents of the initial header and index available to the VFD SWMR reader initialization code.
    d) Initialize the specified VFD to access the target HDF5 file, and instruct it to open that file R/O.
2. On request, it must obtain the current tick and index from the metadata file. Note that this requires either an extension to the VFD interface, or the addition of ad-hoc functions as per the MPIO VFD. As the VFD SWMR Reader VFD will probably be sold as a plugin, an extension to the VFD interface would seem to be required eventually.[25]

[24] If a positive failure is desired when the reader falls behind the writer by more than `max_lag` ticks, we can require the VFD SWMR reader VFD to read the tick from the metadata file header on every metadata read, and fail if the index it is using is more than `max_lag` ticks out of date. Need to decide whether this is worth the overhead in at least some use cases. If so, make it optional.

[25] This feature of VFD interface extension is mentioned as a proposed modification in the VFD Plugin RFC, to be implemented before 2020 (HDF version 1.12).

3. On request, it must use the provided index. Note that this index will be an index that it read from the metadata file.

Where necessary, the added functionality is discussed in greater detail in the following subsections.

### 3.7.1 Selection and Management of the Underlying VFD

Eventually, we will need to define a mechanism for the VFD SWMR Reader VFD to receive and execute instructions specifying the underlying VFD.

In the initial proof of concept implementation, we used hard wired initialization code for the Sec2 VFD and simply passed it the target HDF5 file name. We will retain this until Jake's pluggable VFD feature is ready, and then rework VFD SWMR reader VFD configuration to use his configuration protocol developed for pluggable and stackable VFDs.

### 3.7.2 Index Management

As discussed above in section 3.6 (Metadata Cache Modifications for Reader), shifts from an old index to a new one must be coordinated with evictions from the metadata cache and from the page buffer as well.

To enable this, the VFD SWMR reader VFD must be able to:

1. Report the initial tick, page size, and index immediately after opening the metadata file.
2. Obtain the current tick and index from the metadata file on request.
3. Use the specified index when processing metadata page or multi-page metadata entry read requests.

The internal representations of metadata file indexes are simply arrays of instances of `struct H5FD_vfd_swmr_idx_entry_t`, with the entries sorted in increasing `hdf5_page_offset`. Such an index might be declared as follows:

```
struct H5FD_vfd_swmr_idx_entry_t index[];
```

In principle, the size of the index is variable. However, for the initial implementation, the size of the index was capped by:

`(page size * pages_reserved – header_size – index_overhead[26]) / index_entry_size`

and thus in the initial implementation, index arrays were be allocated to match this size. In the first production version, we will retain this initial allocation, but add code to increase the size of the index should the header and index cease to be adjacent.

Immediately after the VFD SWMR reader VFD opens the metadata file, and the underlying VFD opens the HDF5 file, the reader needs to know the current tick, the page size, and the index. The following functions will support this:

```
herr_t
```

[26] 20 bytes in the index format given above.

```
        H5FD_vfd_swmr_get_page_size(uint32_t *page_size_ptr)
```

If successful, `H5FD_vfd_swmr_get_page_size`() will return the page size read from the metadata file header in *page_size_ptr.

Note that this function will only be called during the file open.

The following function allows access to the current tick and index. It will be used both at file open and in end of tick processing.

```
        herr_t
        H5FD_vfd_swmr_get_tick_and_idx(hbool_t reload_hdr_and_idx,
                                       uint64_t *tick_ptr,
                                       uint32_t index_len_ptr,
                                       struct H5FD_vfd_swmr_idx_entry_t index[])
```

`H5FD_vfd_swmr_get_tick_and_idx()` should proceed as follows:

1. If reload_hdr_and_idx is FALSE, skip this step.
    a. If reload_hdr_and_idx is TRUE, reload the header from the metadata file and check to see if the tick has increased relative to the tick of the reader VFD's local copies of the header and index.
    b. If it has not, set *tick_ptr to the tick read and return.
    c. If the tick has increased, reload the index. Replace the reader VFD's local copies of the header and index with the new versions read.
    d. If the tick has decreased, return an error.
2. Set *tick_ptr equal to the current tick as specified in the reader VFD's local copy of the header.
3. Test to see if *index_len_ptr is less than the value of the Number of Entries field of the reader VFD's local copy. If it is, set *index_len_ptr equal to the Number of Entries field of the reader VFD's local copy, and return.
4. If index is not NULL, copy the reader VFD's local copy of the index into index[], set *index_len_ptr equal to the value of the Number of Entries field of the reader VFD's local copy, and return.

The reload_hdr_and_idx parameter allows the reader VFD to avoid reloading the header and index from the metadata file at file open or if the preceding invocation of the function failed to return the index because the supplied index array was too small.

If the index has not changed, there is nothing to do, and thus the function can simply advise the caller of this fact and return.

In the initial implementation, we let the reader VFD start using new indexes as soon as they are read. However, depending on how we implement optimizations to minimize VFD SWMR reader metadata cache evictions at tick start, it may be necessary to delay use of the new index briefly. If so, we will need a function to set the index in the reader VFD – most likely something along the lines of:

```
herr_t
H5FD_vfd_swmr_set_idx(uint64_t tick, uint32_t index_len,
                      struct H5FD_vfd_swmr_idx_entry_t index[])
```

If so, we will update this document accordingly.

### 3.7.3 VFD Interface Extensions

For the initial proof of concept implementation, we create ad-hoc VFD API calls as per the existing MPIO VFD. We will retain these until Jake's pluggable VFD feature is ready, and then make the necessary additions to the VFD interface.

### 3.7.4 Deltas for NFS

TBD

### 3.7.5 Deltas for Object Stores

TBD

## 3.8 File Open

VFD SWMR has the advantage of making no changes to the HDF5 file, or to the pattern of metadata writes to the HDF5 file.[27] This simplifies matters for the VFD SWMR writer, as its file open processing is limited to initializing some variables, and creating the metadata file.

File open for the VFD SWMR reader is complicated by the fact that both the page buffer and the VFD SWMR reader VFD need to know the page size. As page size is normally stored in a superblock extension message, and this message will frequently be inaccessible without the reader VFD.

This circle is squared by including the page size in the metadata file header, which allows the reader VFD to configure itself without prior access to the superblock extension messages.

### 3.8.1 File Open for the VFD SWMR Writer

On file create in VFD SWMR writer mode, the library must:

- Initialize the VFD SWMR related fields in the associated instance of `H5F_file_t`.

- Allocate and initialize an instance of `H5F_vfd_swmr_eot_queue_entry_t`. In particular, it must:

    - Set `vfd_swmr_writer` to `TRUE`.
    - Set `tick_num` to `1`.[28]

[27] With the exception of the metadata page / multi-page entry writes that must be delayed to avoid message from the future bugs.

[28] Tick 0 is reserved for use as a canonical invalid value. In pages in the page buffer, a value of zero in the `delay_write_until` field is used to indicate that the page (or multi-page metadata entry) may be written immediately.

  - Set `end_of_tick` to the current time plus the tick length.
  - Set `vfd_swmr_file` to point to the instance of `H5F_file_t` of the VFD SWMR file.
- Insert the new instance of `H5F_vfd_swmr_eot_queue_entry_t` into the EOT queue. If it is at the head of the queue, copy its `vfd_swrm_writer` and `end_of_tick` fields into the global variables of the same name.
- Create the metadata file but not write anything to it. Note that it is an error if the metadata file exists prior to file create – if it does, it is possible that we have multiple VFD SWMR writers for the file, and thus the operation should fail.
- Create the log file if requested. If it already exists, it must be truncated.

On file open of an existing HDF5 file in in VFD SWMR writer mode, the library must also:

- Write the header and an empty index to the metadata file

This is necessary to allow the reader immediate access to the existing HDF5 file.

### 3.8.2 File Open for the VFD SWMR Reader

On file open the VFD SWMR reader must:

- Wait until the metadata file exists and contains a header and an index.
- Read the header and index. The index must be saved for comparison with the next index read.
- Configure the underlying VFD (Sec2 for now) and open the target HDF5 file.
- Initialize the VFD SWMR related fields in the associated instance of `H5F_file_t`.
- Allocate and initialize an instance of `H5F_vfd_swmr_eot_queue_entry_t`. In particular, it must:
  - Set `vfd_swmr_writer` to `FALSE`.
  - Set `tick_num` to the current tick read from the metadata file.
  - Set `end_of_tick` to the current time plus the tick length.
  - Set `vfd_swmr_file` to point to the instance of `H5F_file_t` of the VFD SWMR file.
- Insert the new instance of `H5F_vfd_swmr_eot_queue_entry_t` into the EOT queue. If it is at the head of the queue, copy its `vfd_swrm_writer` and `end_of_tick` fields into the global variables of the same name.
- Create the log file if requested. If it already exists, it must be truncated.

## 3.9 End of Tick Functions

The writer and reader end of tick functions are called when the end of tick is detected by the API FUNC ENTER / EXIT macros. These functions were outlined in section 2 above. Now that we have discussed the underlying functionality required to support them, we discuss them again in greater detail

### 3.9.1 Writer End of Tick Function

The writer end of tick function performs the following activities:

1. Flush the metadata cache to the page buffer.

   Metadata pages are flushed to the file as normal unless they exist in the HDF5 file, but not in the metadata file. Such entries must be held for at least max_lag tick before they are flushed so as to provide a consistent view of metadata for the VFD SWMR readers.[29]

   Whether they are flushed or not, copies of all metadata pages or multi-page metadata entries modified in the current tick must be retained in the page buffer until the end of the tick, at which point they may be flushed and/or evicted as normal (with the above proviso).

2. Construct a list of all metadata pages / multi-page metadata entries inserted or modified in the current tick.

3. Allocate space for the entries on this list in the metadata file, and decorate the list with the metadata file offsets, lengths, and checksums.

4. Using this list, and the metadata file index from the previous tick, construct an updated index for the metadata file. In passing, remove entries from the index if the referenced metadata has been written to the HDF5 file and not changed for at least `max_lag` ticks (see above for details).

5. If necessary, allocate space for the new index, and deallocate space for the old index as discussed above. Note that removing the index from the reserved space directly after the metadata file header is a one way trip – once the index grows large enough to force this, the index will not be moved back even if it shrinks.

6. Write the modified metadata pages, multi-page metadata entries, metadata file index, and header to the metadata file as discussed above.

7. Release space in the metadata file used by versions of metadata pages and/or multi-page metadata entries and possibly indexes that have been superseded more than max_lag ticks ago.

   Add the space used by versions of metadata pages and/or multi-page metadata entries (and possibly the index) that were superseded in this tick to the delayed free space release linked list.

[29] Observe that this implies that the page buffer must have access to the metadata file index from the last tick so that it can determine which page / multi-page entry writes must be held for max_lag ticks.

8. Remove the H5F_vfd_swmr_eot_queue_entry_t from the EOT queue, update its end_of_tick and tick_num fields, and re-insert it in the EOT queue. Update the end_of_tick and tick_num globals if the head of the EOT queue has changed. Update the SWMR related fields in the associated instance of H5F_file_t.

9. Resume normal processing.

For efficiency, step 6 above should be managed by a separate thread – however we will not attempt this in the first production version. Note that this optimization will raise dynamically allocated buffer management issues that the current approach avoids.

### 3.9.2 Reader End of Tick Function

The reader end of tick function performs the following activities:

1. Direct the reader VFD to load the current header and tick. If the tick hasn't changed, do nothing and exit.

2. Examine the new index, and determine which pages and/or multi-page entries have been modified since the last time a new index was reloaded. Evict pages from the page buffer and possibly modified entries from the metadata cache as described above.

3. Remove the H5F_vfd_swmr_eot_queue_entry_t from the EOT queue, update its end_of_tick field and set its tick_num field to the value returned by the reader VFD. Re-insert it in the EOT queue and update the end_of_tick and tick_num globals if the head of the EOT queue has changed. Update the SWMR related fields in the associated instance of H5F_file_t.

4. Resume normal processing.

Observe that we do not increment the tick if we don't see a new tick in the metadata file. In such cases, this implies that we will query the metadata file on each API call entry. If this proves to be a problem, we should allow the user to specify a retry delay.

## 3.10 File Flush and Close

The major issue that the VFD SWMR writer has to deal with on file flush or close is the possible need to delay the writes of some metadata pages or multi-page metadata entries. Recall that if a metadata page or multipage metadata entry exists in the HDF5 file and is modified, it must not be written to the HDF5 file until it has appeared in the metadata file index for at least max_lag ticks.

This implies that on HDF5 file flush, the VFD SWMR writer must:

1. Test to see if the page buffer delayed write list is empty. If it is, we are done.
2. Sleep until the end of the current tick.

3. Run the writer end of tick function
4. Goto 1.

Needless to say, this makes the H5Fflush() call very expensive, and something to be avoided in VFD SWMR writer mode. Fortunately, it is hard to see any reason for flushing the HDF5 file in this context.

File close in VFD SWMR writer mode only adds slightly to the overhead of file flush. Here the writer must wait until the HDF5 file is flushed and about to close, and then:

1. Increment the tick
2. Write an empty index to the metadata file. Note that the header must be updated as well.
3. Close and unlink[30] the metadata file

While H5Fclose() is also a potentially expensive operation, we would not expect this to be an issue. If it is a problem, the overhead can be avoided by creating the HDF5 file and not flushing it until file close.[31]

Obviously, the VFD SWMR reader has nothing to do on flush. When the reader closes a file that was opened as a VFD SWMR reader, the VFD SWMR reader VFD must:

- Relay the close to the underlying VFD which accesses the HDF5 file,
- Wait until the close of the HDF5 file is complete, and
- Close the metadata file

Observe that once the VFD SWMR writer and all the readers have closed the HDF5 file, the metadata file will be deleted from the file system.[32]

### 3.11 Logging

The purpose of the log file is to allow us to easily diagnose issues with VFD SWMR. The set of events to be logged will change over time, but will likely include:

- Time of VFD SWMR file open (writer or reader)
- Time at which end of tick is triggered (writer or reader)
- Time required for end of tick processing (writer or reader)
- Size of metadata file index at end of tick (writer only)
- Entries added, deleted, or modified in the index in the past tick (writer only)

[30] Here, unlink refers to the UNIX system call of the same name.

[31] Note that on other than POSIX file systems, this approach may not work due to the possibility that writes may not be strictly ordered. Note also the hidden assumption that no entries have been removed from the index.

[32] For debugging purposes, we should have an option of retaining the metadata file after HDF5 file close.

- Count and total size of metadata pages and/or multi-page metadata entries added to the metadata file at end of tick (writer only)
- Count and total size of metadata pages evicted from the page buffer at end of tick (reader only)
- Count and total size of metadata cache entries evicted from the metadata cache at end of tick. (reader only)
- Time of VFD SWMR file close (writer or reader)

A pared down version of the log file should be available for operational use in determining a safe value for max_lag.

### 3.11.1 Structure of Log File Entries

While the exigencies of implementation will drive the details of the log file, we can specify some structural issues now.

#### *3.11.1.1 Format of Log File Entries*

Each log file entry should have the following syntax:

```
<log_file_entry> ::= <time_stamp> <entry_type_tag> <body> ‘\n’

<time_stamp> ::= time at which log entry was created – format TBD

<entry_type_tag> ::= “FILE_OPEN” | “FILE_CLOSE” | “END_OF_TICK” |
                     “EOT_PROCESSING_TIME” | …

<body> ::= text string
```

As indicated above, the exact format of the time stamp is TBD, with the following constraints.

- The overhead of obtaining the current time should be minimized.
- If practical, the time stamp should offer at least 0.1 second resolution.

The entry type tags are used to indicate the type of log entry, allowing us to grep for series of entries of interest. Note that it must be easy to add new entry types.

The body is simply a text string provided as part of the log entry.

#### *3.11.1.2 Log Entry Reporting Function*

The VFD SWMR log entry function should have a signature along the lines of the following:

```
void H5F_post_vfd_swrm_log_entry(H5F_file_t f; int entry_type_code, char * body);
```

where:

`f` is a pointer to the instance of `H5F_file_t` of the file that has been opened for either VFD SWMR write or read.

`entry_type_code` is an integer specifying the type of the log message, and indexes into an array of strings containing the entry type tags.

`body` is an arbitrary string.

If the target instance to `H5F_file_t` doesn't refer to a file that is open for VFD SWMR read or write, or if the log file is undefined, the function is a NO-OP.

Otherwise, write the log file entry to the log file, using the indicated entry type, or "UNDEFINED" if the `entry_type_code` parameter is out of range.

For the release version, there should be a switch allowing us to suppress all but a small subset of the log file entries based on the `entry_type_code`. However, this is not necessary for the first cut.

# 4 Implementation Details

TBD

# 5 Testing

TODO: Update this section to reflect the current status of the regression test code.

In the proof of concept implementation, we re-used the existing SWMR tests to avoid the costs of writing a proper test suite for VFD SWMR. This was reasonable in that context, and made it possible to develop a near complete proof of concept version of VFD SWMR during phase 1. However, a comprehensive test suite is needed both to validate the initial production version, and to verify continued correct behavior as the initial version is optimized, and as the existing version of SWMR is removed from the library.

Testing for VFD SWMR falls into three categories – unit, integration, and performance testing. Each of these is discussed in turn in the following sections.

## 5.1 Unit Tests

Unit tests are intended to verify correct behavior of major components of VFD SWMR. This section will have to be expanded in the future, but for now we simply list the components and behaviors to be verified. Expect this list to expand.

- New page buffer
  - Correct behavior in non-VFD SWMR mode (existing test suite is weak)
  - Correct operation of tick list
  - Correct operation of delayed write list
  - Page and multi-page metadata entry invalidations work correctly
- Metadata file creation and update
  - Correct free space management
  - Correct management of index when index size exceeds space allocated for it
  - Correct data and index writes
- VFD SWMR reader VFD
  - Correct management of torn writes

  - Correct management of header and index with mismatched tick
  - Correct index lookups
  - Correct reads and pass throughs
  - Correct management of underlying VFDs
- Metadata cache modifications to support VFD SWMR reader
  - Correct behavior of page index
- EOT Queue
  - Correct entry insertion and deletion, and update of `vfd_swmr_writer` and `end_of_tick` globals
  - End of tick functions triggered as expected (do this via logging function)
- Flush in VFD SWMR writer mode
  - Verify correct delays to allow delayed write list to drain. Use log file for this?

## 5.2 Integration Tests

Integration testing verifies that the major components discussed above interact with each other and the existing HDF5 library to yield the desired functionality – in this case, full SWMR. In this case, full SWMR implies that the full capabilities of the HDF5 library function as expected while operating in VFD SWMR writer mode, and that:

- the VFD SWMR specific API calls perform as expected,
- multiple VFD SWMR files perform as expected,
- the files generated are continuously readable by other processes that have opened them in VFD SWMR reader mode, and
- changes to metadata (and raw data if the `flush_raw_data` flag is set) are visible to the reader in no more than 3 ticks.

Conceptually, this is a daunting task, as at least in principle, it requires us to take most of the existing test suite and refactor it so that the writes take place on the VFD SWMR writer, and that the data written is verified on both the VFD SWMR reader and writer.

Fortunately, the architecture of VFD SWMR simplifies this greatly, as makes no functional changes on the writer above the level of the metadata cache and page buffer. Thus it should be sufficient to exercise all the metadata cache clients on the VFD SWMR writer, and verify that the expected changes appear on the VFD SWMR readers within three ticks. The existing create and verify zoo functions in the cache image tests should provide a good starting point.

TODO: Flesh out the details of the integration tests. An incomplete list of features to be covered follows:

- VFD SWMR specific API calls
- variable length data

- shared object header messages
- dataset creation, extension, contraction, and deletion
- all dataset types
- all indexing methods
- group creation, entry insertion and deletion, and group deletion. Be sure to cover the phase shifts in internal representation used in the latest version groups.

## 5.3 Performance Tests

The objective of the performance tests is to compare VFD SWMR performance and existing SWMR, and to support performance regression tests. Consider the following tests:

- Create 10,000 extensible datasets and round robin through them 10,000 times adding small amounts of data on each pass. Measure the following metrics:
    - Total elapsed time for both creation and round robin phases
    - Min, max, and average time from data write to visibility.
    - If practical, min, max, and average time for dataset creation and write.
- Create ten n x 1000 x 1000 dataset of int32, where the first dimension is extensible and the chunk size is 1 x 1000 x 1000. Round robin between the data sets writing a 1 x 1000 x 1000 plane on each pass, with the first dimension starting at 0 and increasing by 1 on each pass. Measure the following metrics:
    - Write speed
    - Min, max, and average time from data write to visibility.

# 6 Recommendation

Review the current version of the VFD SWMR design and point out any issues discovered.

Assuming that no fatal objections are raised, implement changes / expansions of the initial implementation to construct the initial production version.

Flesh out the design and implement the needed test suite.

# Acknowledgements

Development of the initial sketch design for VFD SWMR was supported by ECP (further ID?).

Subsequent design work and implementation supported by a DOE SBIR grant (further ID?).

# Revision History

*July 30, 2018:* Version 1 circulated for comment.

*August 3, 2019:* Version 2 updated in preparation for phase 2 of the SBIR. Major updates include:

- Added design details for supporting multiple files opened in VFD (reader or writer) mode.
- Added enable / disable end of tick API calls
- Added design overview of the new page buffer.
- Updated metadata file index management to support floating indexes when index size exceed metadata file allocated for it, and removal of entries from the metadata file index if their referents have been written to the HDF5 file and not changes for more than `max_lag` ticks.
- Corrected discussion of flush and close in VFD SWMR writer mode.
- Wrote first cut of testing section.
- Addressed reviewer comments.

*September 2, 2019* Version 3 circulated for external review and comment.

*October 27, 2019* Corrected error in pseudo code in section 3.3.2

Version 4 circulated for external review and comment.

*September 16, 2020* Incomplete updates in preparation for the Friendly User release. In particular, Section 3 has become dated, and should be brought into conformance with the code prior to production release.