RFC: Shared Chunk Cache Design

John Mainzer (john.mainzer@lifeboat.llc),
Elena Pourmal (elena.pourmal@lifeboat.llc)
Lifeboat, LLC

Introduction of the structured chunk storage to support sparse data in HDF5 [1, 2] requires rework of some components of the HDF5 library, specifically, rework of chunk cache: the chunk cache has to support I/O on chunks that may have two or more sections with different kinds of data stored in each section. Since we will be rewriting this component, it is our opportunity to address not only I/O for regular and structured chunks but also to address known deficiencies of the current chunk cache implementation.

The purpose of this document is to summarize new requirements for HDF5 chunk cache, sketch out the design of a shared chunk cache, and to solicit input from the HDF5 developers' community before finalizing its design.

The new chunk cache implementation that supports sparse storage and to addresses known deficiencies will be contributed to the open source HDF5 software maintained by The HDF Group.

1 Introduction

Lifeboat, LLC has been working on designing sparse data storage in HDF5.

Sparse data is common in many scientific disciplines and experiments. Several examples are discussed in [3], including High Energy Physics and Neutron and X-ray Scattering. In these examples, only 0.1% to 10% of gathered data is of interest. HDF5, due to its proven track record and flexibility, remains the data format of choice. As the amount of data produced continues to grow due to higher instrument resolution and higher sampling rates, there is a demand for efficient management of sparse data in HDF5.

In HDF5, problem-sized data is stored in multidimensional arrays of elements of a given type. Currently, the HDF5 library requires that *all* elements are defined with user-supplied values or fill values, i.e., it treats data as "dense", mapping *each* data element to storage during I/O operations. HDF5 chunking and per-dataset compression help to optimize the storage of sparse data. However, there are several obvious disadvantages to applying "dense storage" to sparse data: the location of the user-supplied data is not explicitly represented; when read into memory (and after decompression), may result in a huge memory footprint. Therefore, a different approach for sparse data management is needed. The proposed approach was first prototyped in [3], and outlined next to provide some context to the requirements discussed in section 2.

As it is impractical to hold the entire sparse dataset in memory, we break the sparse dataset into user-specified, regular, n-dimensional hyper-rectangles. A sparse chunk is a hyper-rectangle endowed with an HDF5 selection, which represents all defined (i.e., user-supplied) entries in its domain. This way, each sparse chunk contains a selection (data coordinates) and associated data. This approach allows us to store only data of interest and to operate on sparse chunks using existing HDF5 facilities for serialization and deserialization, and for constructing partial and parallel I/O on sparse data.

We generalized the idea of sparse chunk described above and designed a new storage layout called Structured Chunk. Like a regular chunk, a structured chunk will be used to store the values contained in some n-dimensional rectangular volume in a dataset. However, unlike regular chunks, it will be composed of two or more sections, which in combination, will describe the values contained in the volume. For sparse chunk that contains data of fixed-size datatype (e.g., HDF5 numeric type) sparse chunk will have two sections: one section to store encoded coordinates of the defined values and another section to store the defined values themselves. When storing sparse data of variable-length datatype sparse chunk will have three sections: a section to store coordinates, a section with the defined values, and a section with the pointers to the defined values. As with the regular chunked layout the structured chunk layout will support partial and parallel I/O, and data filtering that can be applied to different sections of the structured chunk.

We have already outlined the extensions to the HDF5 file format [1] and public APIs [2] to structured chunk storage. Now we are in the process of designing the HDF5 library changes keeping in mind that in the future, structured chunk can be used not only for storing sparse data of any datatype including variable-length datatype (e.g., strings) but also for storing dense variable-length data and non-homogenous data arrays (e.g., arrays in which each element has its own datatype).

Since chunk cache is one of the most important components of the library and has to work with both dense and structured chunks, we started our design for the HDF5 library changes with designing new chunk cache.

The following sections discuss motivations for new design, outline the requirements, sketch the design and the internal API provided to the HDF5 library.

The objective of this document is to solicit input from the HDF5 developers' community in the hope that design errors and oversights will be discovered sooner rather than later.

A single thread version of the new chunk cache implementation will be contributed to the open source HDF5 software maintained by The HDF Group.

2 Chunk Cache Requirements

This section summarizes motivations for chunk cache re-implementation and outlines[1] requirements for new implementation.

There are two major drivers for re-implementing chunk cache: support for new type of chunk – structured chunk, and known deficiencies of the current cache design and implementation. The next subsections outline the requirements for each driver.

2.1 Support for Structured Chunk

The addition of support for sparse data (and therefore structured chunks) will require significant awareness of structured chunks on the part of the chunk cache – in particular, it must:

1. *Cache un-encoded versions of sections of structured chunks.* The initial application of this ability will be the selections associated with structured chunks used to store sparse datasets. In the future we will need to add un-encoded versions of pointers into variable-length data heaps for sparse and dense variable-length datasets and tables of types used in non-homogeneous data sets.
2. *Be able to adapt to changing structured chunk section sizes*. For example, in the sparse chunk case, both the encoded selection and the fixed data sections will change size as entries in the structured chunks are defined and undefined. Similarly, for variable length data, heap size will change as variable length data is added, deleted, or modified.
3. *Be able to perform operations on individual sections of a structured chunk both on load and in preparation for write.* For example, for sparse data, the selection section will have to be decoded on read, and encoded prior to write. Similarly, it will be useful to defrag heap sections just prior to filtering and writing to disk if the percentage of unused space exceeds some user defined value.

[1] This document is work-in-progress and listed requirements are preliminary.

4. *Run different filter pipelines on the different sections of structured chunks on read.* Similarly, on write, run different filter pipelines on the various sections of the structured chunk, and assemble the resulting buffers into the *on-disk* image of the structured chunk on write.
5. *Manage checksums on individual sections of structured chunks where required.* This is necessary, as sections of structured chunks that indicate where raw data is to be found (i.e., the encoded selections used in sparse data) are in effect metadata, whose correctness must be verified to prevent buffer overflows and related run time errors.
6. *Be able to read and cache individual sections of structured chunks.* A possible application is to cache only the selection sections of structured chunks in sparse datasets, so as to allow efficient determination of the locations of defined data across the entire dataset. Whether this will prove a useful optimization remains to be seen, but if we want to explore it, the capability should be designed into the chunk cache to begin with. The facility could also be used to support reads / writes of individual pieces of data without loading the full structured chunk in the un-filtered case – as is currently done with contiguous and unfiltered chunked data sets.

2.2 Addressing current deficiencies

The existing implementation has a number of problematic design decisions. One of them is the decision to create one chunk cache per open chunked data set[2]. When large numbers of datasets must be held open, this results in a large allocation of RAM to the chunk caches, the vast majority of which is un-used at any point in time. Very often this leads to a poor performance due to memory swapping.

Further, by default, the library creates a 1 MiB chunk cache for each opened chunked data set. This results in thrashing if the dataset in question was created with a chunk size greater than 1 MiB and data filtering enabled, and if the I/O request is not aligned with logical chunk boundaries.

Finally, the indexing method used by the current chunk cache uses a hash table for indexing. While this is common enough, the hash table handles collisions by evicting the pre-existing entry. To minimize the chances of pathological situations resulting from this design decision, the hash table is made larger than would otherwise be necessary. While this seems to control the problem, it would be good to resolve it completely.

The following requirements should be fulfilled to address current chunk cache deficiencies regardless of the chunk type.

1. *Shared chunk cache:* To avoid the chunk cache size explosion when large numbers of datasets are open simultaneously, share the chunk cache between all open datasets. Whether we do this on a per file basis, or across all open datasets in all open files is an open question[3]. Given

[2] It seems that the original intent was to have one shared chunk cache per file. However, performance was poor – reportedly due to I/O operations of any size on a given dataset pushing all other datasets out of the chunk cache. The one chunk cache per open dataset design was chosen to resolve this issue. While this problem should also be manageable with space reservations and an appropriate replacement policy, this tale points out yet another potential pitfall.

[3] Whether sharing the chunk cache between all open datasets in all open files is useful is unknown at this point. While we don't see any obvious downsides in the single thread case, multiple threads operating on multiple files may raise resource allocation issues between the threads. Further, if not all files are on the same file system and thus have different latencies, there may be additional performance and/or fairness issues.

the complexities of a global chunk cache, we currently lean towards a per file chunk cache – but the idea should be explored.

2. *Track which chunks belong to which datasets.* We need this to facilitate flushing individual datasets, both on user command and on close. It also facilitates replacement policies based on datasets as well as individual chunks.
3. *Support for minimum chunk cache space allocations on a per dataset basis.*
4. *Support for alternate replacement policies.* While the initial replacement policy will almost certainly be some variation on LRU, it is easy to come up with scenarios where alternate policies would be useful. Support for this should be designed in.
5. *Support for adaptive chunk cache resizing.* The metadata cache has had the ability to adjust its size to match the current working set size within user specified limits and time frames since HDF5 version 1.8. Given the success of this feature, and how the working set size of the chunk cache can vary based on access patterns, adding a similar facility to the chunk cache seems obvious.
6. *Support for background thread(s) prefetching chunks, or preparing dirty chunks for flush when they become likely candidates for flush and/or eviction.* Given the cost of filtering large chunks, this should provide a significant performance boost when there is available compute to support it.
7. *Multithread support.* While the version of the chunk cache for the existing HDF5 library will have to be single thread, the Lifeboat version will have to be multi-thread to fit into the H5+ product.
8. *Make chunk cache available for parallel I/O.*
   To do this, move some or all support for collective I/O on chunked datasets with variable-size chunks to the chunk cache and/or the chunk format layer. Similar capability already exists in HDF5 to allow (collective only) writes to filtered chunked datasets in the parallel build. As the algorithm is essentially the same for any dataset with variable size chunks, generalizing it and moving it into the chunk cache and/or the chunk format layer would allow this code to be shared between all such types of chunked datasets – present and future.

3 Conceptual Overview

As the current objective is to outline the general structure of the proposed new chunk cache, and circulate it for review, this design sketch will be very high level, with many details omitted.

This section is divided into four subsections. The first describes the main data structures, the second how these data structures are used in common operations, and the third discusses the problems to be addressed in designing the initial replacement policy code.

The fourth section is a collection of design notes and suggestions that, while not particularly relevant at this stage of the design work, will become more useful as the design is fleshed out.

3.1 Overview of Data Structures

Since the objective is to support multi-thread operations, the data structures in the shared chunk cache must be multi-thread safe, even if the initial implementation will be single thread. While some locking

on the chunk level will almost certainly be necessary, experience with retrofitting multi-thread on the HDF5 library suggests that lock-free data structures should be used to the extent possible.

The shared chunk cache will require two main indexes, or three if it is shared across multiple files. While it is far from settled, these will probably be lock-free hash tables.

3.1.1 Chunk Index

The chunk index will be used to store all chunks currently in the shared chunk cache. Entries in the chunk index will be structures containing, among other things,

- the identifier (ID) of the host file,
- the ID of the host dataset,
- the coordinates of the chunk,
- the chunk ID, and
- pointers to buffers containing the sections of the target chunk.

The chunk ID presents a problem. In the higher levels of the chunked data set code, chunks are identified by their coordinates in the dataset. As these coordinates can contain up to 31 integers, this identifier is not suitable for use as an index. Instead, we will probably use a single integer computed from the dimensions of the dataset and the coordinates of the chunk in question.

The obvious problem with this is that the dimensions of the dataset is one of the inputs – as a result, chunk IDs may change if the dimensions of the dataset are modified. Fortunately, this happens infrequently, but when it does, chunk IDs for the target dataset will have to be recomputed. If the chunk ID changes, the associated chunk must be removed from the chunk index and re-inserted with the new ID.

This chunk ID is used to index the chunks, with the dataset ID and the file ID used to resolve any chunk ID collisions.

3.1.2 Dataset Index

The dataset index will be used to store information on all datasets that have been registered with the shared chunk cache. Entries in the dataset index will be structures containing, among other things:

- the ID of the host file,
- the ID of the dataset,
- the number of chunks from the dataset currently resident in the chunk cache,
- the root of a linked list of all resident chunks.
- Pointers to the chunk specific
  - encode,
  - decode,
  - read,
  - write,
  - defined values, and
  - erase

callbacks. These callbacks are used by the shared chunk cache to perform the indicated operations on individual chunks that are currently resident in the cache.

Entries in the dataset index are indexed by the dataset ID, which will probably be the offset of the dataset object header in the file. The file ID will be used to resolve any collisions.

3.1.3 File Index

The file index will be used to store information about files that are currently registered with the shared chunk cache. If we go the one shared chunk cache per file route, it will only have one entry, and will probably not be implemented as an index.

Each entry in the index will contain the ID of the file, and any data necessary to perform I/O on that file.

3.2 Common Operations

At a high level, this section outlines the sequence of actions that the shared chunk cache will take to perform a read, a write, and a chunk flush.

3.2.1 Read

On a read call, the shared chunk cache will receive a vector of

(<file id>, <dataset id>, <chunk id>, <memory selection>, <memory buffer>, <file selection>)

6-tuples describing the desired reads on a per chunk basis. At the conceptual level, the cache will iterate through this vector performing the following operations:

- Lookup the <chunk id> in the chunk index. If it doesn't exist,
  - look up its address and extent in the index[4] maintained by the dataset,
  - load the on-disk image of the chunk into a buffer,
  - call the decode callback to convert the on-disk image into the in-memory representation,
  - insert this in-memory representation into the chunk index, and
  - insert the chunk in the list of in cache chunks maintained for the host dataset.

  Note that this operation will typically trigger one or more chunk flushes and/or evictions.
- Call the read callback to read the data indicated by the <file selection> into the <memory buffer> as indicated by the memory selection.
- Update the replacement policy data structures for the activity on this chunk.

In practice, processing will probably be a bit different from this. For example, to optimize I/O, the chunk cache will likely scan through the vector and perform the desired I/O on all resident chunks first, then, cache space permitting it will read the missing chunks in a single vector read for efficiency. This will usually trigger one or more chunk evictions, with any flushes of dirty chunks handled in a single vector write if possible.

[4]This is the B-Tree, extensible array, or fixed array maintained for the dataset that maps the chunk to its location and extent in file. This index is not to be confused with the chunk index mentioned above.

Note also that this cycle of operation omits a variety of case specific optimizations. For example, in the unfiltered dense chunk case, it may be cheaper to read the data directly from file instead of loading the entire chunk.

3.2.2 Write

On a write call, the shared chunk cache will receive a vector of

`(<file_id>, <dataset_id>, <chunk_id>, <memory_selection>, <memory buffer>, <file_selection>)`

6-tuples describing the desired writes on a per chunk basis. At the conceptual level, the cache will iterate through this vector performing the following operations:

- Lookup the `<chunk id>` in the chunk index. If it doesn't exist,
  - look up its address and extent in the index maintained by the dataset,
  - load the on-disk image of the chunk into a buffer,
  - call the decode callback to convert the on-disk image into the in-memory representation,
  - insert this in-memory representation into the chunk index, and
  - insert the chunk into the list of in cache chunks maintained for the host dataset.

  Note that this operation will typically trigger one or more chunk flushes and/or evictions.
- Call the write callback to write the data indicated by the `<memory selection>` from the `<memory buffer>` into the in-memory image of the chunk as indicated by the `<file selection>`. Note that the size of the chunk's in memory footprint may change as a result of this write. If so, the chunk's entry in the chunk index and the current size of the cache must be updated accordingly.
- Update the replacement policy data structures for the activity on this chunk.

In practice, processing will probably be a bit different from this. For example, to optimize I/O, the chunk cache will likely scan through the vector and perform the desired I/O on all resident chunks first, then, cache space permitting it will read the missing chunks in a single vector read for efficiency, insert them into the chunk cache and perform the desired writes. This will usually trigger one or more chunk evictions, with any flushes of dirty chunks handled in a single vector write.

As before, this cycle of operation omits a variety of possible optimizations. For example, chunk writes that overwrite the entire chunk can be written directly to file, bypassing the shared chunk cache, as can all writes to unfiltered chunked datasets.

3.2.3 Chunk Flush

There are several ways that the flush of a chunk can be triggered. While all of them must be detailed in future versions of this document, for now it seems sufficient to describe the processing required for a chunk flush. At the conceptual level, the processing proceeds as follows:

- Call the dataset specific encode callback on the target chunk to obtain the on disk image of the chunk, the size of this image, and the section offsets in the structured chunk case.
  - If this image size has changed, call the memory manager to allocate a new location in the file for the chunk, write it to its new location, update the index maintained by the dataset for the new location, extent of the chunk and (in the structured chunk case) its

section offsets, and then release the file space allocated to the old version of the chunk image.

  - If the image size hasn't changed, just overwrite the old image with the new and update the index maintained by the dataset to reflect any changes in section offsets.

- Update the target chunk's chunk index entry and the replacement policy to reflect the fact that the entry is now clean.

If, in addition, the target chunk is filtered, its entry in the index maintained by the dataset will have to be updated to reflect this as well.

3.3 Replacement Policy Consideration

As mentioned above in a footnote, the current chunk cache was originally intended to be shared between all open datasets in the target file. It was converted to its current one cache per open dataset configuration due to poor performance, reportedly caused by dataset operations on one dataset evicting partially read or written chunks of other datasets, which then had to be re-loaded.

This cautionary tale points out that developing replacement policies for the shared chunk cache will not be trivial. Thus, the initial objective should be to replicate the effects of the current chunk cache through per dataset allocations, and preferences for retaining partially written or read chunks. This done, we should examine use cases, and develop replacement policies tailored to them.

While they should be verified, two use cases come to mind.

The first of these is simple read or write of entire datasets. In this case, a replacement policy that prioritized retention of partially read or written chunks would seem to be a reasonable starting point to be refined through experimentation. Note, however, that in the write case, this idea will not be applicable to sparse datasets.

The second is random reads of datasets as seen in machine learning applications. This is a difficult case, as this randomness makes the working set size equal to the dataset size. Here, making the cache as large as possible and using either a LRU or random replacement policy would seem to be to be an obvious starting point.

3.4 Design Notes

This section is a catchall for design points that, while irrelevant to the current version of this document, must be addressed as this document is refined.

Multi-Thread Support

While the initial version of the shared chunk cache will have to be single thread in the sense of processing only one request at a time, even this version can benefit from thread pools to allow some level of parallelism executing I/O requests on chunks that are resident in cache, in decoding chunks that have just been read, and in encoding chunks prior to write. Thus, as mentioned above, the internals of the shared chunk cache must be multi-thread safe.

The final version should be able to process multiple requests at a time – but this raises two additional issues.

First, while disjoint I/O requests can operate in parallel, I/O request that overlap must be processed in receipt order. This suggests that I/O requests must be received in some sequential order, examined for overlaps with any requests currently awaiting execution, and then queued as necessary to avoid interleaved execution. This problem appears in the I/O concentrators used to implement sub-filing, but it may be more difficult in the context of the shared chunk cache.

Note also that this issue applies to datasets that would not normally pass through the chunk cache. Thus, for example, it may be convenient to pass contiguous dataset I/O through the chunk cache to maintain serialization of I/O requests.

Second, while disjoint I/O requests may execute in parallel without fear of file corruption, they do raise resource allocation issues. Specifically, the shared chunk cache should attempt to minimize the number and maximize the size of I/O requests that it passes down to the VFD layer. This requires memory to stage the I/O requests, and in the absence of same, these requests will have to be broken up. Thus, available cache space will have to be a consideration in releasing operations for execution. At least initially, we will have to use a first come/first served policy, but we should explore relaxing this to maximize throughput without compromising correctness.

Other points

Combining caches: It has been suggested that the shared chunk cache and the metadata cache should be combined. Given the very different behaviors of cached raw data and metadata and the potential for ID collisions, this seems likely to introduce a great deal of complexity to no purpose. That said, we will need a multi-thread safe metadata cache. Thus, the shared chunk cache should be developed with sharing code with a metadata cache in mind.

Adding mandatory checksums: It has also been suggested that all chunks should be "checksumed" for security reasons. While this is a policy question that we will not address here, it is needed for some sections of structured chunks. If convenient, a general capability for all sections of all chunks should be implemented.

Implementation stages: It has been observed that with some effort, the chunked dataset code could be modified to use either the existing chunk cache, or the shared chunk cache. This idea should be explored, as it would allow the shared chunk cache to be introduced dataset type by dataset type instead of all at once – reducing the stress involved in its implementation.

4 API Sketch

This section sketches out the API of the shared chunk cache as seen by other components of the HDF5 library. The objective is to provide the necessary services in a manner that minimizes the task of integrating the shared chunk cache.

This sketch is intentionally high level pending detailed review of the existing chunk cache interface. The objective in this iteration is provide enough detail for this design sketch, and to allow useful review to identify misconceptions and oversights. In addition, reviewers are invited to point out potential integration issues.

The remainder of this section lists major API calls, and outlines their functionality and required inputs.

4.1 Initialize Chunk Cache

The purpose of this call is to stand up the shared chunk cache with its initial configuration.

Configuration data will include:

- Replacement Policy: ID of the replacement policy to be used, along with any necessary configuration data.
- Adaptive Resize Policy: ID of the adaptive cache size adjustment policy, along with the minimum, maximum, and initial size, and any associated configuration data.
- Internal Thread Pool Configuration: Number of threads to allocate to the internal thread pool, the tasks they are assigned to, and any associated configuration data. At present, three sets of tasks are considered:
  - Preparing “cold” entries for flush to disk, and
  - Prefetching.
  - Executing I/O operations on chunks currently in cache.
- Target File: In the one file per shared chunk cache case, we must tell the shared chunk cache where to direct its I/O requests as part of configuration. Need to decide whether the shared chunk cache should talk to the top of the VFD stack directly, or if its I/O requests should pass through the page buffer as well.
  If we choose to share the chunk cache between all open files, this file configuration data will be passed to the shared chunk cache via the Register File call discussed below.

4.2 Configure Chunk Cache

Get and set the configuration an exist chunk cache. Parameters are the same as the Initialize Chunk Cache call above, save that the target file(s) will either be fixed, or will only be modifiable via the file calls below.

4.3 Discard Chunk Cache

Shut down the target chunk cache, and return its memory to the heap.
Ideally, any remaining registered datasets or files would cause this call to fail. However, this preference may have to bow to the realities of the HDF5 library.

4.4 Register File

In the one shared chunk cache for all open files, add the specified file to the list of files that the shared chunk cache will perform I/O on. Note that the file must be registered before any datasets in the file are registered.

Data required for this call includes:

- ID associated with this file, which must be constant for the length of the file open, and must be supplied with all calls involving this file.
- Reference to the VFD stack used to access this file (possibly with page buffer on top).

4.5 Flush File

Flush all dirty chunks associated with the specified file to disk.

4.6 Un-Register File

For all chunks associated with the specified file, flush them if dirty, evict them, and remove the file from the list of files currently supported by the shared chunk cache.

Need to decide how to handle any registered data sets located in the target file. The obvious solution is to flag an error if any such data sets exist. However, the details of the shutdown code may make it advisable to un-register any such data sets in passing.

4.7 Register Dataset

Configure the shared chunk cache to cache and perform I/O on chunks from the specified dataset.

Data required includes:

- ID associated with the containing file, if we go with a single chunk cache shared across multiple files.
- ID associated with the dataset. This ID must be constant for the lifetime of the containing file open, since it is possible for the dataset to be opened and closed repeatedly, and any chunks still in the shared chunk cache from a previous open/close must be associated with the new open.
- Number of sections in chunks in the datasets – call this value `num_sect`. This value will be 1 for all existing chunked datasets, 2 for sparse datasets without variable length data or dense datasets with variable length data (when re-implemented) and 3 for sparse datasets with variable length data. While no other values for `num_sect` are currently under consideration, the implementation must allow for other values in the future.
- Pointer to the decode callback. This callback accepts:
  - A vector of void pointer of length `num_sect`. Each void pointer points to a buffer containing an image of the associated section of the target chunk.
  - A vector of `size_t` containing the sizes of the above buffers

  and returns:
  - A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections.
  - A vector of `size_t` containing the sizes of the above in memory representations.

  Generally speaking, this callback translates the on-disk representations of the specified chunk sections into their in-memory representations. As currently contemplated, this callback must;
  - Run the supplied on-disk images of the chunk section through filters if defined and applicable in this case.
  - Compute and verify checksums on sections images as required.
  - Decode sections as required
  - Return pointers to the resulting in memory data structures, along with the in-memory sizes.

  The particulars of this function will vary depending on the host dataset.

Thus, for example, it will be a NO-OP for unfiltered, dense, chunked data sets as currently implemented. For sparse datasets with variable length data, the selection section will be run through the filter pipeline if defined and appropriate, the checksum of the resulting buffer will be verified, the resulting buffer decoded into an in-memory representation of the selection specifying the defined values in the chunk, and the address of this selection will be returned. The remaining sections will be run through filters and have their checksums verified if required, and then be returned without further modification.[5]

- Pointer to the encode callback. This callback accepts:
  - A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections.
  - A vector of `size_t` containing the sizes of the above in memory representations.

  And returns:
  - A vector of void pointer of length `num_sect`. Each void pointer points to a buffer containing an image of the associated section in its on-disk representation.
  - A vector of `size_t` containing the sizes of the above buffers.

  Generally speaking, this callback translates the in-memory representations of the specified chunk sections into their on-disk representations. As currently contemplated, this callback must:
  - Compact sections if required
  - Encode sections as required
  - Compute and append checksums to section images as required,
  - Run the supplied on-disk images of the structured chunk sections through filters if defined and applicable in this case.
  - Return pointers to the resulting buffers, along with their sizes in bytes.

  The particulars of this function will vary depending on the host dataset. Thus, for example, it will be a NO-OP for unfiltered, dense, chunked data sets as currently implemented. For sparse datasets with variable length data, the selection section will be encoded, the resulting buffer's checksum will be computed and appended to the buffer, and then run through filters if defined and applicable. The heap section will be checked for excessive waste space, and compacted if appropriate[6], and then run through the filter pipeline if required. The remaining fixed length data section will be run through filters and check summed if required, and then be returned without further modification.

- Pointer to the read callback

[5]At present, the chunk cache is responsible for running chunks through filter pipelines if specified. While making the decode callback responsible for this will facilitate a cleaner shared chunk cache design, the effort required for this structural change may be excessive – in which case responsibility for running chunks through filter pipelines will remain in the shared chunk cache.

[6]Note that compaction of the heap will usually quire modification of the associated offset length pairs in the fixed length data section.

The purpose of the read callback is to read the values specified by the supplied selection from the specified chunk and store them in the supplied buffer as indicated by the supplied memory space selection. The parameters passed to the read callback include:

- A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections.
- A vector of `size_t` containing the sizes of the above in memory representations.
- A selection indicating what data is to be read from the target chunk.
- A buffer and associated selection indicating where data read from the chunk is to be stored.
- A pointer to a float, in which the fraction of the chunk read is returned.

- Pointer to the write callback
  The purpose of the write callback is to write the values supplied to the specified chunk in the specified locations. While it can't happen with existing chunks, observe that with structured chunks, it is possible for the size of a chunk to change as the result of a write. In this context, even if the size of a chunk sections does not change, it may be necessary to allocate new buffers for some or all sections. The parameters passed to the write callback include:
  - A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections.
  - A vector of `size_t` containing the sizes of the above in memory representations.
  - A pointer to a vector of void pointers of length `num_sect`, whose values are initialized to `NULL`. If the write requires allocation of a new buffer for any section of the chunk, its address will be returned in the appropriate cell of this vector.
  - A pointer to a vector of `size_t`, whose values are initialized to zero. If a write requires allocation or reallocation of a chunk section buffer, the size of the new or reallocated buffer is stored in the associated entry in this vector of `size_t`.
  - A buffer and associated selection indicating where the data to be written to the chunk is located.
  - A selection indicating where the indicated data is to be written to the target chunk.
  - A pointer to a float, in which the fraction of the chunk written is returned.
- Pointer to the defined values callback
  The purpose of the defined values callback is to support the construction of a selection of defined values in a dataset, possibly intersected with another selection on the dataset.
  The defined values callback intersects the supplied selection on the indicated chunk with defined values in that chunk and returns the resulting selection. Obviously[7], this is a no-op on anything other than sparse datasets, as the return value is simply the input value.[8]
  Parameters passed to the callback include:

[7]Or perhaps not. One can argue the utility of a callback for dense chunks that would examine the target chunk, construct a selection of the non-fill value entries, and return the intersection of this selection with the supplied selection. This would be slow, but it would provide a mechanism for repacking dense datasets which mostly contain the fill value as sparse datasets.
[8]While it is beyond the scope of this document, there may be some utility in a callback that would scan the target chunk, construct a selection of values that meet some criteria, and return it.

- A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections[9].
- A vector of `size_t` containing the sizes of the above in memory representations.
- The input selection on the target chunk.
- The output selection, which is the intersection of the input selection, with the selection of defined values in the chunk.

- Pointer to the erase callback
  The purpose of the erase callback is to support the removal of values from a sparse dataset[10].
  The function accepts a selection of entries in the chunk to be deleted from the chunk. Conceptually, this selection is intersected with the selection of defined values in the chunk, and the resulting selection is subtracted from the defined values selection. Observe that this operation will typically require creation of new in memory representations of the defined values and fixed length data sections of the structured chunk. If present, it may also be necessary to compact the variable length data heap – although the current plan is to do this in the serialize callback.
  Parameters passed to the callback include:
  - A vector of void pointer of length `num_sect`, containing pointers to the in-memory representations of the associated structured chunk sections[11].
  - A vector of `size_t` containing the sizes of the above in memory representations.
  - A pointer to a vector of void pointers of length `num_sect`, whose values are initialized to `NULL`. If the erase requires allocation of a new buffer for any section of the chunk, its address will be returned in the appropriate cell of this vector.
  - A pointer to a vector of `size_t`, whose values are initialized to zero. If the erase requires allocation or reallocation of a chunk section buffer, the size of the new or reallocated buffer is stored in the associated entry in this vector of `size_t`.
  - An input selection of entries to be removed from the target chunk.

4.8 Un-Register Dataset

What to do with any remaining associated chunks in the shared chunk cache when a dataset is un-registered raises a number of questions. Ideally, those chunks would remain in the cache until pushed out via normal operations on the off chance that the dataset will be re-opened and re-registered. While

[9]As currently contemplated, only the first pointer in this vector will be used. However, as we don't know how we will used the structured chunk concept in the future, it seems prudent to include all data that may be of use. Note that the notion of evicting some sections of a structured chunk without evicting the whole thing has been floated. If we go this route, we may wish to include an "insufficient data" return from the callback, forcing the reload of some or all the missing sections.

[10]The notion of applying this operation to dense datasets has discussed – the suggestion being that the cells indicated by the selection should be overwritten with the fill value.

[11]As currently contemplated, only the first pointer in this vector will be used. However, as we don't know how we will used the structured chunk concept in the future, it seems prudent to include all data that may be of use. Note that the notion of evicting some sections of a structured chunk without evicting the whole thing has been floated. If we go this route, we may wish to include an "insufficient data" return from the callback, forcing the reload of some or all the missing sections.

this will require retaining a private copy of the data from the dataset registration until the last such chunk has been flushed and evicted, this seems doable.

On the other hand, this seems to be a departure from existing practice. Must decide what to do here.

4.9 Read Chunks

At the conceptual level, the read chunks call should accept a vector of

`(<file id>, <dataset id>, <chunk id>, <memory selection>, <memory buffer>, <file selection>)`

6-tuples, and performs the indicated reads from the indicated chunks into the indicated memory locations. For each entry in this vector, the chunk cache must load the chunk if necessary, and then call the read callback discussed above to perform the read.

Chunk cache space permitting, the necessary reads from file (if any) should be packaged into as small a number of vector reads as possible. At least to a cursory review, this seems to be similar to what is already going on in `H5D__chunk_read()`, at least in the multi-dataset / selection I/O case. If so, this should facilitate integration.

Looking at the elements of the 6-tuple in greater detail:

- `<file id>`: ID indicating the containing file of the remaining elements of the 6-tuple. Needless to say, this ID is not needed if we elect a one shared cache per file design.
- `<dataset id>`: ID indicating the containing dataset. This dataset must be currently registered with the shared chunk cache.
- `<chunk id>`: ID assigned to the target chunk. Ideally, this ID must remain constant across datasets opens and closes for the duration of the file open. As discussed elsewhere, this may not be practical as it will probably be necessary to re-assign chunk IDs and re-index any resident chunks whenever the dimensions of the host dataset are modified.
- `<memory selection>`: Selection indicating where data read from the target chunk is to be stored in the memory buffer.
- `<memory buffer>`: Buffer that receives data read from the target chunk.
- `<file selection>`: Selection on the target chunk indicating what data is to be read from the chunk.

4.10 Write Chunks

At the conceptual level, the write chunks call should accept a vector of

`(<file id>, <dataset id>, <chunk id>, <memory selection>, <memory buffer>, <file selection>)`

6-tuples, and performs the indicated writes to the indicated chunks from the indicated memory locations. For each entry in this vector, the chunk cache must load the chunk if necessary[12], and then call the write callback discussed above in section 4.7 to perform the write to the chunk.

Chunk cache space permitting, the necessary writes to file should be packaged into as small a number of vector writes as possible. At least to a cursory review, this seems to be similar to what is already

[12]On write, a chunk need not be loaded if either it has not yet been allocated, or if the write will completely overwrite it.

going on in `H5D__chunk_write()`, at least in the multi-dataset / selection I/O case. If so, this should facilitate integration.

The elements of the 6-tuple are essentially identical to the read chunk case.

In addition to the vector of 6-tuples, it may be useful to add control flags – it particular, it may be useful to have a write through flag.

5 Final Recommendation for Chunk Cache Implementation in HDF5

To be added after discussions with the HDF5 community.

Acknowledgment

This work is supported by the U.S. Department of Energy, Office of Science under Award number DE-SC0023583 for SBIR project “Supporting Sparse Data in HDF5”.

References

1. John Mainzer, Elena Pourmal, “RFC: File Format Changes for Enabling Sparse Storage in HDF5”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-File-Format_Sparse_Storage_Changes-2023-07-17.pdf
2. John Mainzer, Elena Pourmal, “RFC: Programming Model to Support Sparse Data in HDF5”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-Model-API-Sparse-2023-07-18.pdf
3. J. Mainzer *et al*., "Sparse Data Management in HDF5," *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*, Denver, CO, USA, 2019, pp. 20-25, doi: 10.1109/XLOOP49562.2019.00009.

Revision History

January 30, 2024	Version 1 was created for internal review. This document contains requirements published in “New Requirements for HDF5 Chunk Cache” https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-Chunk-Cache-Req-2023-12-18.pdf
January 30, 2024	Version 2 was created for posting on GitHub. Section 3.2.3 was added in this version.