RFC: Shared Chunk Cache Design

Clay Carper clay.carper@lifeboat.llc
John Mainzer john.mainzer@lifeboat.llc

This document outlines the design of the Shared Chunk Cache (SCC). Initially, SCC will support I/O operations for structured chunks—a newly introduced HDF5 storage paradigm for sparse and variable-length data. In the future, it may replace the current chunk cache implementation, extending its functionality to support dense chunks as well. The cache is also designed to enable muti-threaded access in the future.

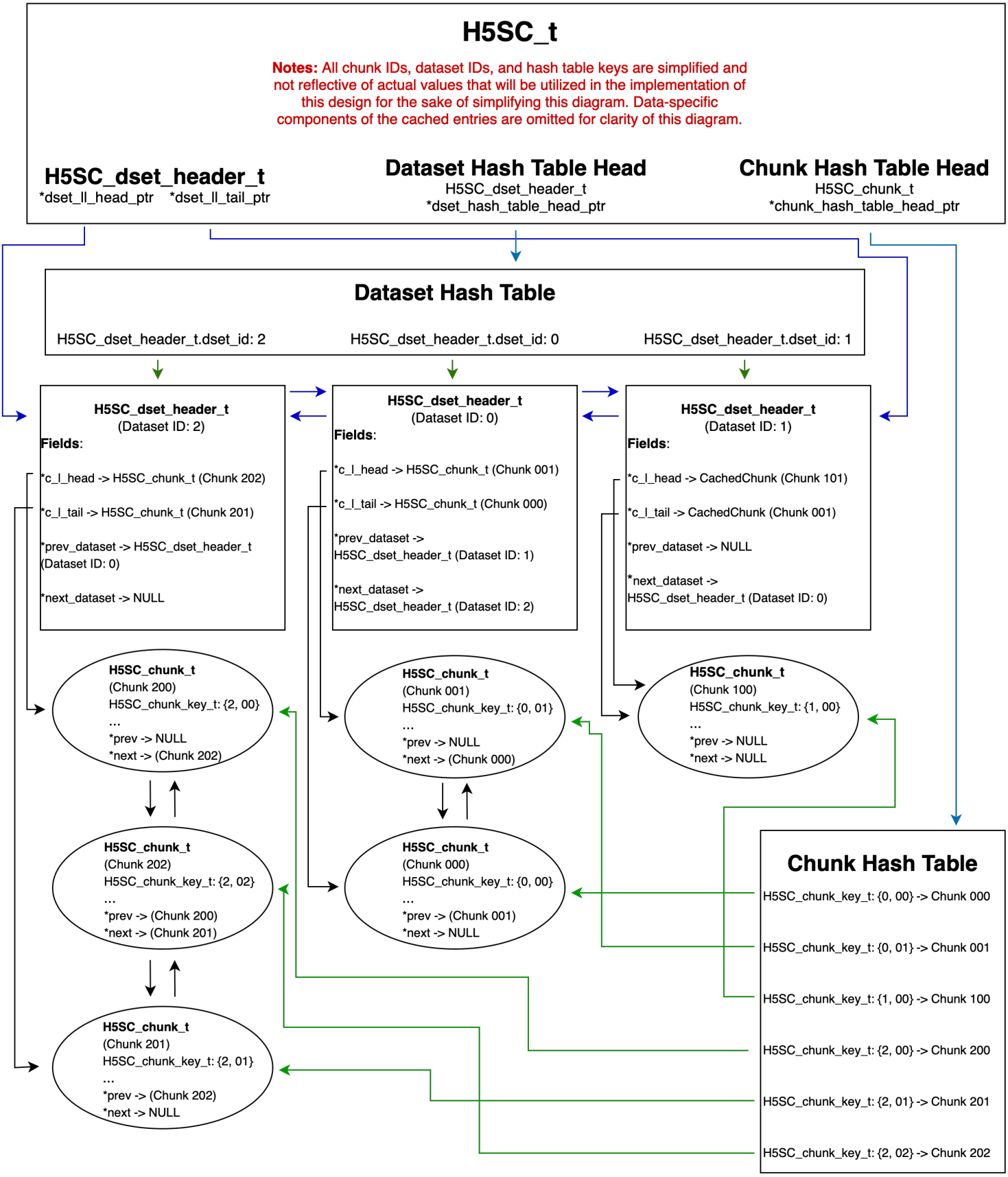
Current document is work in progress and will be updated as we proceed with the Version 1 implementation as described in the document.

1 Introduction

This document outlines the design for the Shared Chunk Cache (SCC). The new cache not only provides support for structured chunks, a new data storage introduced to HDF5, but serves as an opportunity to address additional requirements [1] and known limitations of the existing chunk cache. For example, a single cache will be utilized across all datasets in a single HDF5 file, enabling the SCC to utilize a minimalistic hash table, while also utilizing portions of memory that would have otherwise been inaccessible due to the nature of the previously utilized cache-per-dataset approach. The remainder of this document will focus on the design of the SCC, including the proposed structures, how they satisfy the design requirements, and how various procedures will utilize these components.

2 Overview of Structures

To support and inform the underlying design decisions, this document will first focus on the underlying data structures which have been developed for the initial single-thread-enabled version of the SCC. Additionally, considerations for future multi-thread support outlined in section 5 have been influential in this development. Further, the diagram below provides an illustration of the structures discussed in this document.



2.1. `H5SC_chunk_key_t` – Chunk Indexing

```
/****************************************************************************
 *
 * Structure: H5SC_chunk_key_t
 *
 * Info
 *
 *   The structured stores a unique 128-bit key for a chunk. The key is derived
 *   from the dataset's object header address in the file and the chunk's serialized
 *   logical coordinates using an LSB-first bit interleaving schema.
 *
 * Fields
 *
 *    uint64_t low_half  - The bottom 64-bits of the 128-bit integer key.
 *
 *    uint64_t high_half - The top 64-bits of the 128-bit integer key.
 *
 *    Example: Assume 64-bit dataset object header address = 6₁₀ or …0110₂,
 *             and 64-bit chunk logical coordinate = 5₁₀ or …0101₂.
 *             LSB-first bit interleaving yields to 0,1,1,0,1,1 → 01101100…₂
 *             with remaining zeros omitted for clarity. The result of
 *             the operation is stored in the two 64-bit integer fields
 *             high_half = 01101100…₂; low_half = 00000000…₂;
 *
 ****************************************************************************/

typedef struct H5SC_chunk_key_t {
    uint64_t high_half;
    uint64_t low_half;
} H5SC_chunk_key_t;
```

Design Notes

This section is reserved and will be updated as we proceed with implementation.

2.2. `H5SC_chunk_t` – Individual Cache Entry Structure

```
/****************************************************************************
 *
 * Structure: H5SC_chunk_t
 *
 * Info
 *
 *    The structure represents an individual chunk entry in SCC; it is an entry
 *    in the chunk hash table.
 *
 *    The structure stores the chunk key, pointers to in-memory and on disk chunk
 *    data, chunk size, state flags, and pointers necessary to track the chunk as a
 *    member of the associated dataset's chunk LRU list. A hash handle is also
 *    included to make the structure hashable.
 *
 * Fields
 *
 *    H5SC_chunk_key_t data_key - The unique key associated with the cached
```

```
 *          chunk. It is primarily used for chunk lookups through the chunk-
 *          focused hash table and operations that require internal chunk index
 *          updates.
 *
 *    void *cached_chunk_buf_ptr – The pointer to the decoded, in-memory, chunk
 *          data buffer. This field is Null if no in-cache buffer is available.
 *
 *    void *disk_chunk_buf_ptr – The pointer to the encoded, on disk chunk data
 *          buffer. This field is Null when the chunk is newly created or if there
 *          is no on disk data associated with this chunk.
 *
 *    size_t cached_chunk_size – The size of the in-memory data buffer after
 *          decoding the chunk and the application of any required filters. This
 *          field is updated when the in-memory data is altered. The default
 *          value is set to 0.
 *
 *    size_t disk_chunk_size – The size of the on disk data buffer prior to
 *          applying filters or decoding. It is obtained using the
 *          H5D__struct_chunk_lookup callback. The default value is set to 0.
 *
 *    size_t chunk_counter – A priority counter primarily used with eviction
 *          policies. It is set to 0 by default. In the future, it may be used to
 *          identify eviction candidates based on access history or other encoded
 *          states. For example, during eviction candidate selection, if the
 *          partial_io flag is set to True, this counter would be incremented to a
 *          value of 1 to indicate that it has been flagged for a second pass
 *          through the LRU list.
 *
 *    bool dirty_flag – A flag to indicate when an in-memory chunk buffer has been
 *          modified. A value of True indicates a chunk buffer is dirty. This flag
 *          is set when a chunk is created/modified/resized. Set to False by
 *          default.
 *
 *    bool partial_io – A flag to indicate if a partial read or write operation
 *          has been done on this chunk (i.e., the intersection of the selection
 *          describing this chunk and the selection on the dataset resulted in
 *          less than the whole chunk being selected). It is used to track I/O
 *          states and inform eviction policies. This field is set to false by
 *          default.
 *
 *    struct H5SC_chunk_t *prev_ptr – The pointer to the previous node in the
 *          dataset's LRU list. It is Null if this chunk is at the head of the LRU
 *          list or if the LRU list has a single element.
 *
 *    struct H5SC_chunk_t *next_ptr – The pointer to the next node in the
 *          dataset's LRU list. It is Null if this chunk is at the tail of the LRU
 *          list or if the LRU list has a single element.
 *
 *    UT_hash_handle hval_chunk – A hash handle is required for this structure to
 *          be hashable by UTHash; it is used to track this chunk in the SCC chunk
 *          hash table. Set to Null by default.
 *
 ********************************************************************************/

typedef struct H5SC_chunk_t {
      H5SC_chunk_key_t  data_key;
      void              *cached_chunk_buf_ptr;
      void              *disk_chunk_buf_ptr;
```

```
      size_t            cached_chunk_size;
      size_t            disk_chunk_size;
      size_t            chunk_counter;
      bool              dirty_flag;
      bool              partial_IO;
      H5SC_chunk_t     *prev_ptr;
      H5SC_chunk_t     *next_ptr;
      UT_hash_handle    hval_chunk;
} H5SC_chunk_t;
```

Design Notes

This section is reserved and will be updated as we proceed with implementation.

2.3. `H5SC_dset_header_t` – Dataset-Based Chunk Tracking

```
/******************************************************************************
 *
 * Structure: H5SC_dset_header_t
 *
 * Info
 *
 *    The header structure for a dataset with data held in the shared chunk cache.
 *    It tracks the dataset identity (represented by the dataset object header
 *    address), the minimum amount of chunk data that should be maintained in this
 *    dataset (in bytes), accounts for the total size of cached chunks. The
 *    structure contains pointers to the head and tail of the LRU list for the
 *    cached chunks in this dataset, pointers to the next and previous datasets in
 *    the dataset LRU list, and pointers which are necessary for eviction
 *    management.
 *
 * Fields
 *
 *    haddr_t dset_addr -  The 64-bit address of the object header for this
 *          dataset. This field is used as the key for the dataset hash table.
 *          It is set to 0 by default.
 *
 *    size_t min_dset_size - Represents the minimum number of bytes to
 *          remain cached for this dataset under typical memory constraints. The
 *          default size is 10 MB and is user-configurable.
 *
 *    size_t curr_dset_size - The current amount of chunk data, in bytes
 *          associated with this dataset (i.e., is a sum of chunk sizes). This
 *          quantity should be updated when a chunk is added, removed, or resized.
 *          This field contributes to total cache utilization tracked in H5SC_t.
 *          It is set to 0 by default.
 *
 *    bool resize_in_progress - A flag used to indicate that a resize operation is
 *          being conducted on this dataset. A value of True is set if dataset is
 *          being resized. While true, any internal SCC
 *          procedures and any I/O requests for this dataset should be interrupted
 *          until the resize operation is completed. It is set to false by
 *          default.
 *
 *    bool evict_exhausted - A flag used to indicate that this dataset has
 *          nominated as many eviction candidates as it is able to and should not
 *          be overlooked for future eviction candidate selections during this I/O
 *          cycle. During eviction candidate selection, this field is set to True
```

```
 *          if H5SC_chunk_t elements have been removed and the min_dset_size has
 *          been reached. When this field is True, this dataset header is removed
 *          from the dataset LRU list and is added to an exhausted dataset list
 *          through utilizing the associated pointers (*next_exhausted_ptr and
 *          *prev_exhausted_ptr). This field is set to False by default.
 *
 *    H5SC_chunk_t *lru_head_ptr – The pointer to the head of this dataset's chunk
 *          LRU list (indicating it is the most recently used chunk). Set to
 *          Null by default, when the chunk LRU list is empty, or when the chunk
 *          LRU list contains a single element.
 *
 *    H5SC_chunk_t *lru_tail_ptr – The pointer to the tail of this dataset's chunk
 *          LRU list (indicating it is the least recently used chunk). Set to Null
 *          by default, when the chunk LRU list is empty, or when the chunk LRU
 *          list contains a single element.
 *
 *    H5SC_dset_header_t *next_dset_ptr – The pointer to the next dataset in the
 *          dataset LRU list. Set to Null by default, when this header is at the
 *          tail of the dataset LRU list, or when the list has a single element.
 *
 *    H5SC_dset_header_t *prev_dset_ptr – The pointer to the previous dataset in
 *          the dataset LRU list. Set to Null by default, when this header is at
 *          the head of the dataset LRU list, or when the list has a single
 *          element.
 *
 *    H5SC_dset_header_t *next_exhaustd_ptr; - The pointer to the next dataset in
 *          the exhausted dataset linked list. Set to Null by default, when this
 *          dataset is at the exhausted list tail, or if the exhausted list has a
 *          single element.
 *
 *    H5SC_dset_header_t *prev_exhausted_ptr; - The pointer to the previous
 *          dataset in the exhausted dataset linked list. Set to Null by default,
 *          when this dataset is at the exhausted list head, or if the exhausted
 *          list has a single element.
 *
 *    UT_hash_handle dset_hval – A hash handle is required for this structure to
 *          be hashable by UTHash; it is used to track this dataset in the SCC
 *          dataset hash table. Set to Null by default.
*
 ********************************************************************************/

typedef struct H5SC_dset_header_t {
      haddr_t             dset_addr;
      size_t              curr_dset_size;
      bool                resize_in_progress;
      bool                evict_exhausted;
      H5SC_chunk_t       *lru_head_ptr;
      H5SC_chunk_t       *lru_tail_ptr;
      H5SC_dset_header_t *next_dset_ptr;
      H5SC_dset_header_t *prev_dset_ptr;
      H5SC_dset_header_t *next_exhaustd_ptr;
      H5SC_dset_header_t *prev_exhausted_ptr;
      UT_hash_handle      dset_hval;
} H5SC_dset_header_t;
```

Design Notes

This section is reserved and will be updated as we proceed with implementation.

2.4. `H5SC_t` – Per-File Global Cache Management

```
/****************************************************************************
 *
 * Structure: H5SC_t
 *
 * Info
 *
 *    The primary structure used for shared chunk cache management. Provides
 *          fields for maintaining access to the head and tail of the dataset LRU
 *          list, tracking the active and quiescent cache memory usage, and the
 *          pointers necessary to access the hash tables used for chunk and
 *          dataset lookups.
 *
 * Fields
 *
 *    size_t SCC_quiescent_size - The amount of quiescent data, in bytes, present
 *          within the cache when there are no I/O operation being performed on
 *          cached data. This field is computed using the sum of the
 *          curr_dset_size field from each dataset currently in the SCC. This
 *          field is updated whenever a chunk is added, removed, or resized while
 *          present within the SCC. A value of 0 indicates the cache is empty. A
 *          value of 0 is set by default when this structure is created.
 *
 *    size_t SCC_active_size - The amount of active data, in bytes, allocated to
 *          the SCC when an I/O request is being processed. This field is an
 *          integer-multiple of the current SCC_quiescent_size. The multiplier is
 *          user configurable and is set to be 2x the size of the quiescent limit
 *          by default to accommodate the application of filters or other
 *          operations that often cause chunks to exceed estimated sizes. Once an
 *          I/O request is completed, this field is set to 0.
 *
 *    H5SC_dset_header_t *dset_lru_head_ptr - The pointer to the head of the
 *          dataset LRU list (indicating it is the most recently used dataset).
 *          Set to be Null by default, when the dataset LRU list is empty, or when
 *          the dataset LRU list contains a single element.
 *
 *    H5SC_dset_header_t *dset_lru_tail_ptr - The pointer to the tail of the
 *          dataset LRU list (indicating it is the least recently used dataset).
 *          Set to be Null by default, when the dataset LRU list is empty, or when
 *          the dataset LRU list contains a single element.
 *
 *    H5SC_chunk_t *chunk_hash_table_head_ptr - The pointer to the head of the SCC
 *          chunk hash table. Set to Null when the hash table is empty or by
 *          default. This pointer will only change when the first element of this
 *          hash table is added or removed.
 *
 *    H5SC_dset_header_t *dset_hash_table_head_ptr - The pointer to the head of
 *          the SCC dataset hash table. Set to Null when this hash table is empty
 *          or by default. This pointer will only change when the first element of
 *          this hash table is added or removed.
 *
 ****************************************************************************/

typedef struct H5SC_t {
      size_t              SCC_quiescent_size;
```

```
      size_t               SCC_active_size;
      H5SC_dset_header_t *dset_lru_head_ptr;
      H5SC_dset_header_t *dset_lru_tail_ptr;
      H5SC_chunk_t       *chunk_hash_table_head_ptr;
      H5SC_dset_header_t *dset_hash_table_head_ptr;
} H5SC_t;
```

Design Notes

This section is reserved and will be updated as we proceed with implementation.

3 Theory of Operations

This section provides a conceptual overview of the five primary scenarios involving the cache: opening a file, writing to a file, reading from a file, flushing the cache, and closing a file. Where appropriate, dataset-specific interactions will also be discussed.

3.1. File Open

At the time an HDF5 file is opened, `H5SC_create()` will be called to perform the following actions:

- Allocate and initialize the file specific instance of H5SC_t with default configuration.
- Examine the FAPL for shared chunk cache configuration. If found, overwrite configuration fields in the instance of H5SC_t as specified.

3.2. Dataset Open

The shared chunk cache must be informed whenever a relevant[1] dataset is either opened or created. On receipt of this notification, the shared chunk cache must:

- Allocate and initialize an instance of H5SC_dset_header_t.
- Scan the dataset access property list (DAPL) for non-default dataset configuration data, and if found, copy it into the instance of H5SC_dset_header_t.
- Insert the new instance of H5SC_dset_header_t into the shared chunk cache dataset hash table and dataset LRU list.

3.3. Dataset Flush

When directed, the shared chunk cache must:

- Find the target instance of H5SC_dset_header_t in the dataset hash table in the file specific instance of H5SC_t.
- Scan the chunk LRU list associated with the target instance of H5SC_dset_header_t.

[1] Datasets are supported if their I/O is managed by the shared chunk cache. Initially this means only sparse datasets. However, as the shared chunk cache is integrated into the HDF5 library, all dataset types will become relevant.

- For each instance of H5SC_chunk_t on this list, determine if it is dirty, and flush it to file it if so. Mark the instance of H5SC_chunk_t as clean when done. Note that dirty chunks are looked up on disk and if found, are inserted into the in-file index as needed.
- If directed, evict all instances of H5SC_chunk_t associated with the dataset.

3.4. Dataset Change Extent

Note: This section will be reworked to reflect the details from the H5Dset_extent() section of [2].

When the extent of an open dataset is modified, and I/O to that dataset is routed through the shared chunk cache, the shared chunk cache must be notified. On receipt of this notification, the shared chunk cache must:

- Find the target instance of H5SC_dset_header_t in the dataset hash table in the file specific instance of H5SC_t.
- Remove each instance of H5SC_chunk_t in the associated dataset LRU from the chunk shared chunk cache's chunk hash table.
- Once all instances of H5SC_chunk_t have been removed from the chunk hash table, recalculate each chunks ID and then re-insert it in the chunk hash table under the new ID.

3.5. Dataset Close

The shared chunk cache must be informed whenever a dataset is closed. On receipt of this notification, the shared chunk cache must:

- Find the target instance of H5SC_dset_header_t in the dataset hash table in the file specific instance of H5SC_t.
- Verify that the chunk LRU associated with the target instance of H5SC_dset_header_t is empty. Otherwise, any remaining datasets should be flushed and then have any remaining H5SC_chunk_t elements evicted.
- Unlink the target instance of H5SC_dset_header_t from the dataset hash table and dataset LRU list.
- Free the target instance of H5SC_dset_header_t.

3.6. Writing Data Through the SCC

When writing data through the SCC from an in-memory data buffer to an HDF5 file, `H5SC_write()` is used[2]. When a dataset uses structured chunk storage, the following actions will be taken with respect to the dataset's chunks:

[2] At the time of writing, `H5SC_write` is used to write directly to file in the Version 0 implementation of the cache, and does not convert the chunk buffer into the cache memory as outlined in this section. As a fully functional version of the SCC is implemented, this will be adjusted accordingly.

- Convert the write request into a list of write requests on individual chunks.

- Identify all chunks in this list that are currently resident in the shared chunk cache. Perform the requested writes for these chunks in a single I/O request, decoding the chunks and applying any necessary filter(s), then move each instance of H5SC_chunk_t touched to the head of its per dataset chunk LRU, and its host instance of H5SC_dataset_header_t to the head of the dataset LRU[3].

- Determine the amount of shared chunk cache space that can be made available for this I/O request. This will be a combination of existing free space and evict-able chunks in the shared cache. Call this value S.

- Estimate the total size of the chunks addressed by the remaining chunk write requests using their on-disk size, and divide this list into a minimal set of chunk I/O request sub-lists such that the sum of the sizes of the target chunks in each sub-list is less than S.

- For each such sub-list:

  - Evict sufficient entries from the shared chunk cache to make space for the target chunks in the sub-list. When flushing dirty entries, first construct the on disk images of the target entries, update chunk indexes as necessary for size and location changes[4], and then flush the dirty chunks with a single vector write call.

  - Create or load the target chunks. When loading the on disk images of existing chunks, read the target on disk images with a single vector call, and then decode and apply the appropriate filter(s) to the on disk buffers.

  - Apply the write requests from the sub-list to the now in cache chunks, moving each instance of H5SC_chunk_t touched to the head of its per dataset LRU, and its host instance of H5SC_dataset_header_t to the head of the dataset LRU.

3.7. Reading Data Through the SCC

When reading raw data through the SCC, `H5SC_read()` is used. Similar to the write process, if datasets in the I/O request are supported by the SCC, they are processed in the following way[5]:

- Convert the read request into a list of read requests on individual chunks.

- Identify all chunks in this list that are currently resident in the shared chunk cache. Perform the requested reads on these chunks. Move each instance of H5SC_chunk_t touched to the head

[3] When two or more datasets in a single I/O request, the order in which they are moved to the head of the dataset LRU list is arbitrary, as is the ordering when multiple chunks from a single dataset are moved to the head of the chunk LRU list.

[4] This process will utilize the structured chunk callbacks.

[5] Similarly to the Write to Cache section, Read To Cache From File will utilize the callbacks from `H5Dstruct_chunk.c`, with the details omitted from this section for the time being.

of its per dataset LRU, and its host instance of H5SC_dataset_header_t to the head of the dataset LRU.

- Determine the amount of space that can be made available for this I/O request. This will be a combination of existing free space and evict-able chunks in the shared cache. Call this value S.
- Estimate the total size of the chunks addressed by the remaining per chunk read requests, and divide this list into a minimal set of chunk I/O request sub-lists such that the sum of the sizes of the target chunks in each sub-list is less than S.
- For each such sub-list:
  - Evict sufficient entries from the shared chunk cache to make space for the target chunks in the sub-list. When flushing dirty entries, first look up the chunks, then, if necessary, construct the on disk images of the target entries, update chunk indexes as necessary for size and location changes, and then flush the dirty chunks with a single vector write call.
  - Load the target chunks from file. Do this with a single vector read call, and then decode the on disk images into the in-memory format.
  - Apply the read requests from the sub-list to the now in cache chunks, moving each instance of H5SC_chunk_t touched to the head of its per dataset LRU, and its host instance of H5SC_dataset_header_t to the head of the dataset LRU.

3.8. Flush the Cache

Should all data present in the SCC need to be flushed, `H5SC_flush()` will be utilized. This function will flush all dirty data present in the cache and may also be used to evict members of a dataset. This process will be user-configurable, allowing for per-dataset control of how a cache-wide flush operation should function. This operation is done on a per-dataset basis using the procedure for flushing datasets described above.

3.9. File Close

When an HDF5 file is closed, `H5SC_destroy()` is used to destroy the current SCC. This process does not flush chunks, and frees all data within the SCC using the following procedure:

- Verify that the shared chunk cache is empty.
- Free the file's instance of H5SC_t.

4 Detailed Component Discussions

This section provides descriptions for the various components or operations, such as the hash table and eviction candidate selection.

4.1. Chunk Tracking Through the Hash Table

To maintain consistent, constant-time lookups for chunks, a globally available hash table will be utilized through the use of UTHash. The global hash table utilized by the SCC will rely on the mechanisms provided by UTHash, through the inclusion of a single C header file. Continuing from existing HDF5 paradigms, UTHash utilizes the Jenkins hash function by default, while providing access to additional options should alternate be desirable. For each hash table item, the key, payload (i.e., the values), and the hash handle may be maintained in a single structure. The proposed `H5SC_chunk_key_t` is well suited to serve as a key when using UTHash, and provides a unique, data-focused identification for each individual chunk[6]. The macros for UTHash are well documented [3 and could easily be modified if need be.

4.2. Eviction

Eviction in the SCC follows a two-stage LRU process. The dataset at the tail of the dataset LRU list (pointed to by `dset_ll_tail_ptr` in `H5SC_t`) represents the LRU dataset. Within that dataset, chunks are considered from the tail of its chunk LRU list (`lru_tail_ptr` in `H5SC_dset_header_t`), while working toward head. Eviction candidate selection is done until either the required amount of space has been freed within the SCC or the

The `partial_io` flag provides identification of chunks that were only partially read or written[7]; such chunks will be given a second pass through the cache in favor of fully read or written chunks. Eviction continues within a dataset until its reserved minimum (`min_dset_size`) has been reached, all chunks have been processed, or the required amount of space within the SCC has been freed. After the min_dset_size has been reached with a dataset, it is temporarily removed from the dataset LRU list and added to a list of datasets which have exhausted their ability to nominate further eviction candidates[8].

When a chunk is evicted, it is removed from the dataset's LRU list, unlinked from the global hash table, and its memory freed. Size counters (`curr_dset_size` for the dataset, `SCC_mem_usage` for the SCC) are updated. If further space is required, eviction proceeds to the next dataset in global LRU order.

[6] As a general note, UTHash does not automatically do any verification concerning the uniqueness of a given key. However, the `HASH_FIND` macro is provided by UTHash to allow the user to check whether a key is present within the hash table.

[7] A chunk is considered partially read or written when the selection describing the chunk intersecting the selection on the dataset results in the selected chunk not matching the full chunk selection. This may also occur when the chunk dimension(s) don't evenly divide the dataset dimension(s).

[8] A dataset is considered exhausted when, during eviction, the `min_dset_size` has been achieved, signifying that no additional `H5SC_chunk_t` elements may be removed from this dataset. To avoid potentially repeated processing of this now exhausted dataset, it is temporarily removed from the dataset LRU list and added to a separate linked list. After the eviction process is completed, and datasets on the exhausted list are returned to the dataset LRU list.

4.2.1 Additional Eviction Strategies

The current design supports extension beyond the default two-stage LRU approach. Alternative strategies such as Least Frequently Used (LFU) or Most Recently Used (MRU) can be implemented with minimal changes:

- Frequency-based policies can use the integer counter in each chunk (`chunk_counter`).
- Order-based policies can reuse the doubly linked lists already maintained for datasets and chunks.
- Dataset-level configuration may eventually expose options to select or tune eviction policies.

This flexibility allows new policies to be added without significant restructuring of the SCC.

4.3. Public Configuration Structure

The structure will be modeled after the H5AC_cache_config_t structure available in H5ACpublic.h. The goal of this publicly available structure is to enable users to set dataset-specific minimum retention values, eviction strategies, and global memory limits (active, quiescent, and maximum available). Default values will be included as well, serving as baseline values.

4.4. Version 0 Implementation

The Shared Chunk Cache, Version 0 (V0), is a minimally functional representation of how structured chunk data will be processed through the cache. This initial version focuses on demonstrating the connection between the high-level API (through `H5D__write()`) and the low-level callbacks (which are invoked by the SCC). To maintain a simplistic, verifiable output, a single chunk within a single dataset was initially the sole focus of this development, with the end goal being supporting I/O on multiple chunks within a single dataset. To illustrate proper cache functionality, structured chunk-formatted data would first need to be written to a file and then read back. In V0, the cache simply acts as a pass-through for the data buffer between the high-level API and the low-level callbacks, effectively simulating the action of caching the data (i.e. in this instance is assumed that the cache can only “hold” a single chunk; any additional operations involving the cache would require eviction). This approach serves two primary purposes. First, it was necessary to test both the high-level and low-level code integration, which up to this point, had remained untested (by necessity). Second, the V0 cache would provide the developer further exposure to essential internal HDF5 components that would further inform this redesign. With these goals in mind, the V0 development initially focused on creating, writing, and reading a 1-dimensional dataset with five elements and a single chunk of the same size. For simplicity, the chunk would contain non-fill values at indexes 1, 2, and 3 to provide a quick, consistent point of reference.

To facilitate testing the high- and low-level code integration, the write process was the first to be focused on, through the development of the `H5SC_write()` function. Using chunk information derived from first calling `H5SC__io_info_init()`, two key tasks are done; first the caching action for a dirty chunk is emulated, followed by a simulation of the eviction process. Upon calling `H5SC_write()`, `H5SC__io_info_init()` is called, which processes selections for each chunk in the I/O request. Once this preliminary chunk information is collected, the chunk is looked up on disk (which, by assumption, will result in a failure to find the chunk). Since the chunk is not found on disk, it should be fully

overwritten, which leads to invoking the new chunk callback to initialize a new chunk. After the new chunk is created, the gather memory callback is called to collect the chunk data into a single buffer. Next, the "eviction" process begins. First, the chunk is encoded to ensure that it is transitioned from the in-memory format into the on-disk format. Then, the chunk is inserted into the file indexing using the insert callback function. Finally, the encoded chunk buffer is written to disk using the `H5F_block_write()` function. On-disk data verification was done using the Unix octal dump (od) tool.

After the write process was debugged and verified, the next step was to read the data back into a user buffer through developing the `H5SC_read()` function. Similar to the procedure used to write the initial chunk, `H5SC__io_info_init()` is used to collect necessary information about the requested chunk. Using this information, the chunk is looked up (and this time around, it is expected to be present). If the chunk is found on disk, the chunk read process is initiated. Using the information collected with the lookup callback, `H5F_block_read()` is then invoked to load the on-disk formatted chunk into a buffer. This buffer is then run through the decode callback, which translates the data into the in-memory format. Finally, this buffer is scattered into the available memory using the scatter memory callback.

Once this process was completed for a single chunk in a single dataset, it was then extended to support writes and reads on a single dataset with multiple chunks. During this process, the callbacks were tested further and adjusted to ensure proper functionality when an I/O request necessitates processing multiple chunks.

4.5. Version 1 Implementation

Version 1 (V1) will build on the fundamental components that were developed in V0, while adding the structures outlined in this document and the basic functionality of the SCC. As this development is completed this section will be expanded.

5 Multi-threaded support

This section outlines modifications to support multi-thread operations in SCC and external support code required for multi-thread operation of the shared chunk cache.

5.1. Multi-threaded operations in SCC

The following tasks will be performed to support multi-threading in SCC:

1. Switch to multi-thread safe data structures
    a. Use lock free hash table instead of UThash for the chunk and dataset hash tables,
    b. Replace the doubly linked lists used for the per data set chunk LRUs and the dataset LRU with a lock free equivalent TBD.

2. Modify existing structures to use atomic values where appropriate -- i.e. lengths of and sizes of lists, etc.

3. Code modifications to support the above.

4. While not immediately necessary, for efficiency, we should support concurrent decoding of chunks that have already been read from disk, and encoding of chunks that are about to be

flushed and potentially evicted. We need to implement a thread pool to support this functionality and modify the lower-level callbacks as needed to ensure thread safety.

5.2. Additional code for supporting multi-thread operations in SCC

This sub-section outlines external support code required for multi-thread operation of the shared chunk cache.

Serialization module: When I/O requests are serviced in parallel, we must ensure that only overlapping reads are executed concurrently. Overlapping writes, and any intervening overlapping reads must be serialized. The serialization module examines the stream of incoming I/O requests, and delays passing them to the shared chunk cache as necessary to ensure this. When changing a dataset's extent, we drain all current I/O requests and then marks the dataset as locked until the resizing operation is completed.

Resource allocation module: Large I/O requests can be many times the size of the shared chunk cache. Such I/O requests will require exclusive access to the shared chunk cache if we are to avoid thrashing. More generally, we must estimate I/O request resource requirements and only allow concurrent requests into the shared chunk cache if it has sufficient resources to serve the requests without thrashing. The resource allocation module exists to enforce this.

Multi-thread support in the VFD layer: Concurrent service of multiple I/O requests in the shared chunk cache requires the ability to support concurrent I/O requests in the VFD layer.

Other considerations: Pending multi-thread support in the chunk indices and the metadata cache, we may need to add additional serialization to I/O requests to avoid corruption caused by interleaved modifications to the chunk indexes. This shouldn't be an issue with concurrent reads, however, concurrent writes to the same dataset have potential issues, as do any reads of the dataset concurrent with a write.

The exact potential issues here require further investigation, and it may be that there are no issues not solved by the global mutex.

6 Acknowledgement

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

7 References

1. John Mainzer, Elena Pourmal, “RFC: New Requirements for HDF5 Chunk Cache”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-Chunk-Cache-Req-2023-12-18.pdf.
2. Neil Fortner, John Mainzer, Vailin Choi, Clay Carper, “RFC: Shared Chunk Cache Internal API”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/Shared_Chunk_Cache_API.v11.pdf
3. Hanson, Troy. “Uthash: A Hash Table for c Structures.” *Github.io*, 2012, troydhanson.github.io/uthash/.

Revision History

July 25, 2025:	Version 0 sent for internal review.
July 31, 2025:	Version 0.1 sent for internal review.
August 27, 2025	Version 0.2-0.7 copyediting; 0.6 version added new section of multi-threaded updates.