RFC: Shared Chunk Cache Design

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

This document describes the Shared Chunk Cache design. This new caching mechanism was designed to support structured chunks, a newly introduced format that will support sparse data within HDF5.

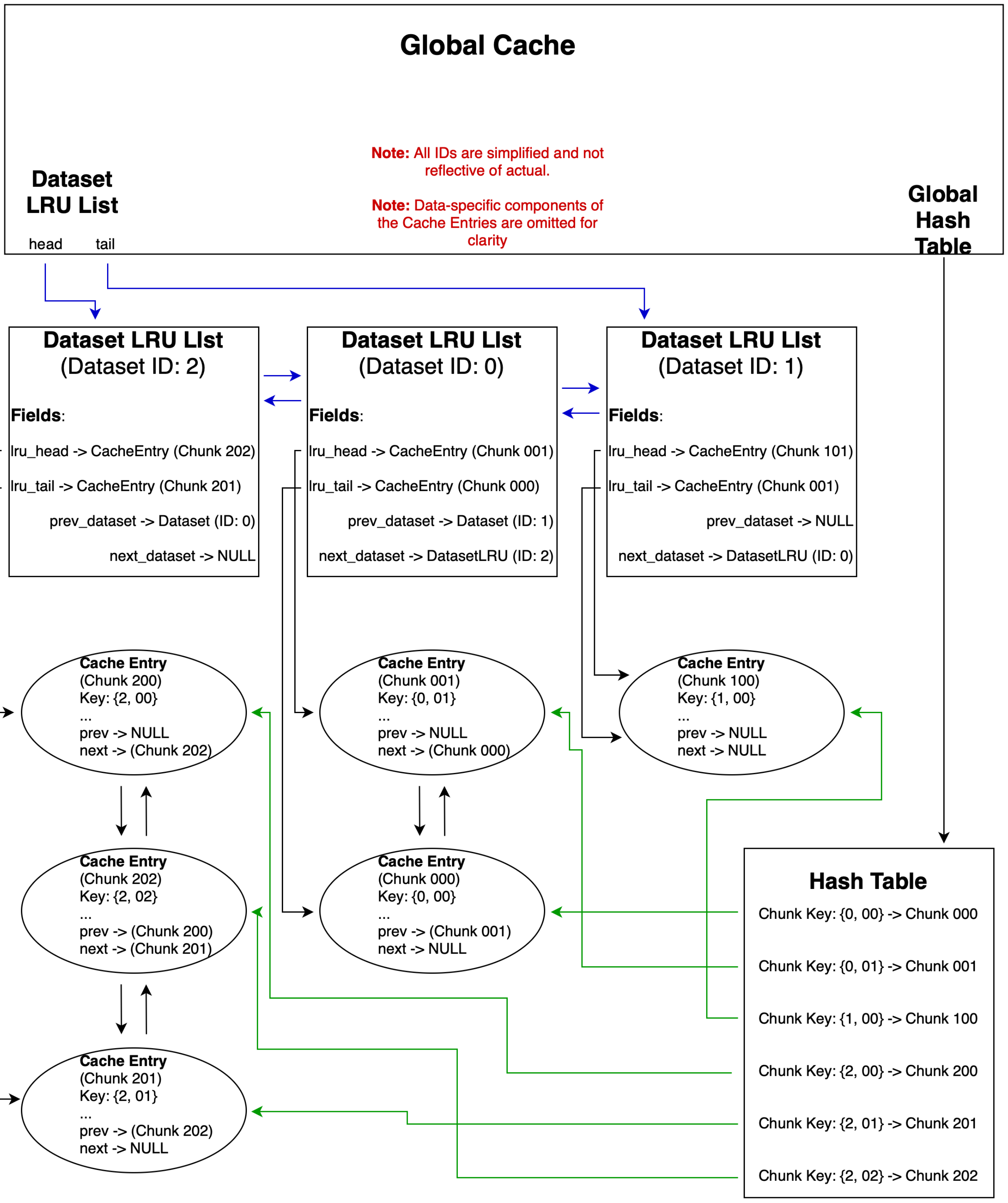
1 Introduction

This document provides the design for the Shared Chunk Cache (SCC). This new cache not only provides support for structured chunks, a new chunk-based data storage introduced to HDF5, but serves as an opportunity to address new requirements [ref to the document in GitHub] 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 have been influential in this development. Further, the diagram below provides an illustration of the structures discussed in this document[1].

[1] The names in the diagram are currently outdated and will be updated shortly.


Global Cache
Note: All IDs are simplified and not reflective of actual.
Note: Data-specific components of the Cache Entries are omitted for clarity
Dataset LRU List
head
tail
Global Hash Table
Dataset LRU LIst (Dataset ID: 2)
Fields:
lru_head -> CacheEntry (Chunk 202)
lru_tail -> CacheEntry (Chunk 201)
prev_dataset -> Dataset (ID: 0)
next_dataset -> NULL
Dataset LRU LIst (Dataset ID: 0)
Fields:
lru_head -> CacheEntry (Chunk 001)
lru_tail -> CacheEntry (Chunk 000)
prev_dataset -> Dataset (ID: 1)
next_dataset -> DatasetLRU (ID: 2)
Dataset LRU LIst (Dataset ID: 1)
Fields:
lru_head -> CacheEntry (Chunk 101)
lru_tail -> CacheEntry (Chunk 001)
prev_dataset -> NULL
next_dataset -> DatasetLRU (ID: 0)
Cache Entry (Chunk 200) Key: {2, 00} ... prev -> NULL next -> (Chunk 202)
Cache Entry (Chunk 202) Key: {2, 02} ... prev -> (Chunk 200) next -> (Chunk 201)
Cache Entry (Chunk 201) Key: {2, 01} ... prev -> (Chunk 202) next -> NULL
Cache Entry (Chunk 001) Key: {0, 01} ... prev -> NULL next -> (Chunk 000)
Cache Entry (Chunk 000) Key: {0, 00} ... prev -> (Chunk 001) next -> NULL
Cache Entry (Chunk 100) Key: {1, 00} ... prev -> NULL next -> NULL
Hash Table
Chunk Key: {0, 00} -> Chunk 000
Chunk Key: {0, 01} -> Chunk 001
Chunk Key: {1, 00} -> Chunk 100
Chunk Key: {2, 00} -> Chunk 200
Chunk Key: {2, 01} -> Chunk 201
Chunk Key: {2, 02} -> Chunk 202

2.1. *ChunkKey* – Chunk Indexing

```
/*****************************************************************************
 *
 * Structure: ChunkKey
 *
 * Info
 *    Stores a unique chunk key derived from the dataset ID and the computed
 *    chunk index.
 *
 * Fields
 *    long long int low_half  - Bottom 64-bits of the 128-bit integer key
 *    long long int high_half - Top 64-bits of the 128-bit integer key
 *
 *****************************************************************************/

typedef struct ChunkKey {
    long long int high_half;
    long long int low_half;
} ChunkKey;
```

The `ChunkKey` is a 128-bit key, consisting of two 64-bit values. This 128-bit key provides a unique identification for each individual chunk across all datasets. A 128-bit key provides plenty of room to support extremely high counts of both datasets and chunked data, while also leaving plenty of space for future development[2]. The `ChunkKey` consists of two values associated with the chunk; the first is the integer identifier (ID) associated with the dataset that the chunk originates from, while the second value is the linear index computed from the offset associated with a given chunk. In this context, offset refers to an array of hsize_t associated with the logical dataset position of the desired chunk. The dataset ID is unique; to speak broadly, once a dataset has been loaded, the integer ID assigned to that dataset will be consistent, so long as the dataset or the file are not closed[3]. However, the chunk ID is not guaranteed to be unique among all data chunks stored in the file. Across two different chunked datasets, each dataset will have a chunk with an offset equivalent to the first chunk (chunk ID 0). Given this limitation, the key associated with each unique chunk must also be unique, while maintaining a derivation that is rooted in existing HDF5 components. Additionally, the `ChunkKey` should provide adequate dispersion for hash table keys, leading to fewer hash collisions, a strongly uniform bucket distribution, and better hash table performance. To accommodate these requirements, each `ChunkKey` is created using the following algorithm. First the dataset ID and chunk offset are each converted from their native HDF5 types into a 64-bit integer Afterward, these two integers are combined using LSB-first bit-level interleaving. For example, if a chunk has a dataset index of 6_{10} and a computed chunk index of 5_{10}, the resulting `ChunkKey` would be $01101100\ldots_2$ (omitting the additional zeros from the underlying 128-bit value). After this interleaving is completed,

[2] For example, support for multiple files or external datasets could be integrated into the ChunkKey structure.

[3] To be more precise, when a dataset is created or loaded, an address of the object header with type haddr_t, is assigned to that dataset. This Unsigned 64-bit integer is a unique and is maintained by the HDF5 free space manager. My understanding on this idea is based on my understanding of object headers and discussions with John.

the resulting 128-bit key will be stored as two 64-bit values in the `ChunkKey` structure. Optimizations may be made to improve performance, though these notions are reserved as future improvements.

2.2. `CachedChunk` – Individual Cache Entry Structure

```
/*****************************************************************************
 *
 * Structure: CachedChunk
 *
 * Info
 *    Represents an individual chunk, including metadata, the chunk's
 *    position in the linked list associated with the dataset, and
 *    information related to this entry in the global hash table.
 *
 * Fields
 *    ChunkKey data_key - Unique key for the cache entry
 *    void *cached_chunk_buf - Pointer to the decoded, in-cache memory formatted
 *          chunk data (NULL if not available)
 *    void *prev_chunk_buf - Pointer to the previous chunk buffer (in-file or in-
 *          memory formatted; NULL if chunk is new, or previous buffer is not
 *          preserved)
 *    size_t cached_size - Size of the cached chunk, computed after
 *          (de)compression/conversion (in-cache memory formatted size)
 *    int chunk_counter - Unsigned integer available for tracking data access.
 *          Set to 0 by default.
 *    bool dirty_flag - Flag to indicate when a chunk has been modified (a value
 *          of True indicates dirty)
 *    bool partial_io - Flag to indicate when a chunk has partially read/written
 *          (a value of True indicates a partial read/write); A partially
 *          read/write occurs when a portion of a chunk is outside the current
 *          extent of a dataset, resulting in an I/O request processing a subset
 *          of the full chunk data. This primarily pertains to traditional dense
 *          chunks; this flag is present to support future feature development.
 *    struct CachedChunk *prev - Pointer to the previous node in the associated
 *          dataset's LRU list
 *    struct CachedChunk *next - Pointer to the next node in the associated
 *          dataset's LRU list
 *    UT_hash_handle hval - Handle for this chunk; used by UTHash to track this
 *          struct in the global hash table
 *
 *
 *****************************************************************************/

typedef struct CachedChunk {
      ChunkKey        data_key;
      void           *cached_chunk_buf;
      void           *prev_chunk_buf;
      size_t          cached_size;
      int             chunk_counter;
      bool            dirty_flag;
      bool            partial_IO;
      CachedChunk    *prev;
      CachedChunk    *next;
      UT_hash_handle  hval;
} CachedChunk;
```

Discussion: This structure provides an in-cache representation of a data chunk, including pointers to the current and previous chunk representation(s)[4], a tag to make the structure a UTHash-able, and other critical components. The key associated with this chunk is stored both to have a reference for which chunk an instance of the chunk represents and for situations where the chunk index needs to be updated[5]. In particular, two boolean values and an integer-based counter provide a per-chunk way of tracking useful chunk states. Determining whether a partial write or read has occurred will be handled within the SCC and as such, will need to be tracked for each chunk with the `partial_IO` field. Further, this field provides support for integrating eviction strategies that dictate eviction candidate selection based on whether a chunk is fully written or read. Similarly, the `dirty_flag` provides a simple way of tracking if a chunk is dirty, that is, the in-cache data is different from the on-disk data[6]. The `chunk_counter` field is present to facilitate eviction strategies that prioritize chunks in various states, allowing chunks with a lower eviction candidate selection priority to be marked as seen. For example, assume that a dirty, fully written or read chunks have a higher priority when selecting eviction candidates when using a pseudo-Least Recently Used (LRU) schema. Since the chunks available within the SCC for a given dataset are present within a doubly linked list, the element at the tail is the first chunk to be considered as an eviction candidate. If this chunk is dirty and `partial_IO` is True, the `chunk_counter` will be incremented by 1, with the chunk then being moved to the head of the LL. In this modified LRU eviction schema, the eviction algorithm would need to maintain an internal counter for what the `chunk_counter` should be on a given pass, with internal logic that allows for chunks with different counters to be prioritized based on the individual `chunk_counter` values. This structure is intended to retain minimalistic chunk information, while maintaining the information necessary to cache functionality.

[4] Depending on the circumstance, there may not be a previous representation of the chunk to maintain a pointer to.

[5] For example, when the chunk index is recomputed due to a change in chunk size, or when a chunk is inserted in such a way that the typical chunk index schema would be disrupted.

[6] A dirty chunk may be a new chunk that is not yet written to file, or a chunk which has had any portion of the data has been altered after being read from file.

2.3. DsetList – Dataset-Based Chunk Tracking

```
/*******************************************************************************
 *
 * Structure: DsetList
 *
 * Info
 *    A structure that contains the head and tail of the linked list associated
 *    with a specific dataset that provides relevant statistics and the pointers
 *    necessary to maintain a per-dataset memory configuration.
 *
 * Fields
 *    size_t dset_id - Unique dataset identifier associated with the appropriate
 *          chunks; derived from the dataset metadata
 *    size_t min_dset_size - Minimum size allocated for this dataset; 10MB is the
 *          default
 *    size_t curr_dset_size - size Current total size of the chunks cached for
 *          this dataset
 *    CachedChunk *chunk_list_head - Pointer to the head of the dataset-specific
 *          linked list (typically the most recently used chunk)
 *    CachedChunk *chunk_list_tail - Pointer to the tail of the dataset-specific
 *          linked list (typically the least recently used chunk)
 *    DsetList *next_dset - Pointer to the next dataset in the global dataset
 *          linked list. This value is NULL if this DsetList is the head element
 *    DsetList *prev_dset - Pointer to the previous dataset in the global dataset
 *          linked list. This value is NULL if this DsetList is the tail element
 *
 *******************************************************************************/

typedef struct DsetList {
      size_t       dset_id;
      size_t       min_dset_size;
      size_t       curr_dset_size;
      CachedChunk *chunk_list_head;
      CachedChunk *chunk_list_tail;
      DsetList    *next_dset;
      DsetList    *prev_dset;
} DsetList;
```

Discussion: The core functionality of this structure serves to maintain a doubly linked list (LL) of chunks associated with an individual dataset, along with useful statistics. To provide the ability to spot-check which LL is associated with a particular dataset, `dset_id` is the dataset ID, which is typically sourced from `H5D_dset_io_info_t` structure. Regarding the statistics, `min_dset_size` is a user-configurable quantity that is set to 10MB by default which serves one primary purpose; it represents the minimum number of bytes that should be held within the cache for chunks in this dataset. A minimum is utilized to ensure that space for critical data is prioritized and maintained within the cache for each individual dataset, while also serving as a simplistic cut-off point for eviction candidate selection[7]. Additionally, the `curr_dset_size` field provides a concise reference for total size of the data held within this linked

[7] For some eviction strategies, the min_dset_size may be ignored based on other criteria such as whether data is partially written or read.

list. This field is referenced by the global cache to maintain an accurate measure of how much data is present within the cache. Both `*chunk_list_head` and `*chunk_list_tail` are references to the head and tail of the LL for this dataset, respectively. These pointers will be used for eviction candidate selection strategies. Finally, the `*next_dset` and `*prev_dset` pointers provide the foundation for the a doubly linked list that maintains an ordering for the datasets present within the SCC.

2.4. GlobalSSC – Per-File Global Cache Management

```
/*******************************************************************************
 *
 * Structure: GlobalSCC
 *
 * Info
 *    Global structure used to manage the shared chunk cache through utilizing a

 *    UTHash-based global hash table for establishing universal chunk lookups and
 *    maintains the linked list used to track the ordering of datasets available
 *    in the SCC.
 *
 * Fields
 *
 *    size_t SCC_mem_max - Global SSC memory limit; user configurable
 *    size_t SCC_mem_min - Global SCC memory minimum; sum of the
 *          min_dset_size for each dataset currently loaded in the SCC
 *    size_t SCC_mem_usage - Current total SCC size (sum) across all datasets
 *          with chunks in the cache; updated whenever a chunk is added or evicted
 *    size_t SCC_active_size - Global size of the active data (if
 *          ((SCC_mem_usage) - (SCC_mem_min)) is positive); approximation of
 *          the current quantity of data that could be evicted under a
 *          traditional LRU process
 *    size_t SCC_quiescent_size - Global size for the quiescent data (if
 *          ((SCC_mem_usage) -(SCC_mem_min)) is non-positive, take the
 *          absolute value)); reflects the approximate amount of data present in
 *          the cache that is at or below the sum of the minimum dataset size of
 *          each loaded dataset)
 *    DsetList *dataset_ll_head - Pointer to the head of the dataset linked lists,
 *          indicating the most recently accessed dataset
 *    DsetList *dataset_ll_tail - Pointer to the tail of the dataset linked lists,
 *          indicating the least recently accessed dataset
 *    CacheEntry *hash_table - Pointer to the head of the hash table data
 *          structure for global chunk lookups
 *
 ******************************************************************************/

typedef struct GlobalSSC {
      size_t      SCC_mem_max;
      size_t      SCC_mem_min;
      size_t      SCC_mem_usage;
      size_t      SCC_active_size;
      size_t      SCC_quiescent_size;
      DsetList   *dset_ll_head;
      DsetList   *dset_ll_tail;
      CacheEntry *hash_table;
} GlobalSSC;
```

Discussion:

In order to have a single, cohesive cache, for an HDF5 file, `GlobalSCC` serves as the top-level structure for this redesign. In particular, this structure contains the fields necessary for enforcing global memory limits, provides pointers to the head and tail of the linked list (LL) containing the `DsetList` structures, and is one of the central components necessary for coordinating eviction candidate selection. In terms of memory management, the maximum, minimum, and current memory usage constraints are each set or managed by this structure. In terms of the maximum amount of memory allocated for the cache across all datasets, `SCC_mem_max` reflects the maximum amount of memory available to the cache. This quantity will be user configurable, with a default value being derived from available system information[8]. Likewise, the `SCC_mem_min` field is the sum of `min_dataset_size` from each dataset currently loaded in the cache. In making this value dynamic, other statistics can be leveraged to provide functionality that would otherwise be restrictive. Similarly, the `SCC_mem_usage` is the sum of the `curr_dset_size` for each dataset present in the cache. Any time a cache operation is done that involves I/O, this quantity will be updated to reflect any change done to a dataset in terms of the amount of data loaded. To accommodate some eviction strategies and provide useful statistics, the `SCC_active_size` and `SCC_quiescent_size` fields are set based on whether the difference of `SCC_mem_usage` and `SCC_mem_min` is positive or not. When this quantity is positive, it is set to the `SCC_active_size`, which provides an approximation of the quantity of data, in bytes, which is likely to be available for eviction[9]. Likewise, when the aforementioned computation is non-positive, the `SCC_quiescent_size`, is set. This field represents the approximate amount of data, in bytes, present in the cache that is at or below the `SCC_mem_min`. Next, pointers to the head and tail of the DsetList LL provides a mechanism for straight forward access to the datasets currently present within the cache. Finally, the `*hash_table` pointer provides a reference to the first entry in the hash table. This value will be `NULL` when the hash table is empty, points to the first entry in the hash table, and only changes when the first element is added or removed.

2.5. Placeholder Subsection Heading

What else?

3 Theory of Operations

This section provides a conceptual overview of the five primary situations that will invoke or involve the cache; when a file is opened, writing to a file, reading from a file, the flushing the cache, and what occurs when a file is closed. Where appropriate, dataset-specific interactions will be discussed. While the SCC will ultimately support vector operations[10], this section will describe the operations as if each chunk is processed as a single request.

[8] The exact mechanism used to derive what the default value will be could also be some multiple of the sum of the dataset minimums, though this notion will need to be explored a bit more.

[9] Currently, there is no plans for this quantity to differentiate between fully read/written chunks and partially read/written chunks. As this development progresses, it may be worth distinguishing between the two. Should this be the case, the dsetList structure will also require minor additions to reflect this functionality.

[10] Vector operations refer to batched I/O requests, allowing for non-contiguous data selections, such as hyperslab selections, point selections, and/or strided access patterns, to be processed in a single call.

3.1. File Open

At the time an HDF5 file is opened, `H5SC_create()` will be called to create a new, empty SCC which support the SCC[11], the following actions will be taken:

- The user configurable parameters for the SCC will be read from publicly available configuration struct
- The SCC is initialized, starting with the creation of the `GlobalSCC` using parameters from the configuration struct or using default values
  - The `SCC_mem_max` field is set based on the public configuration struct
  - `*dset_list_ll_head` and *`dset_list_ll_tail` are set to `NULL` by default
  - The hash table head (`CacheEntry *hash_table`) is set to `NULL` by default
- Once a dataset is created or loaded for this file, the other relevant fields will be updated accordingly

3.2. Write To Cache

When writing data through the SCC from an in-memory data buffer to an HDF5 file, `H5SC_write()` is used[12]. When a dataset supports the SCC, the following actions will be taken with respect to the chunks being cached[13]:

- First, the write request is passed to the SCC
- The size of the request is calculated
  - If the request will fit in the available memory, proceed without conducting eviction candidate selection
  - If not, proceed to eviction candidate selection[14]
- For each chunk in the cache write request, the associated `ChunkKey` is computed

[11] Support for the SCC has typically been determined by assessing whether the field contained within `dset_info[i].dset→shared→layout.sc_ops` returns as `True` or `False`, where `i` is the `i`th dataset in the HDF5 file. See H5Dio.c for an example of this process.

[12] 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.

[13] The details concerning the steps necessary for writing the chunks to a file are omitted from this discussion and are largely handled by the callbacks located in `H5Dstruct_chunk.c`.

[14] In the case where an I/O request is too large to fit in the SCC regardless of the result of eviction candidate selection, the request will be processed in selections that can fit in the available memory.

- The hash table is queried for each `ChunkKey` in the I/O request
    - If a `ChunkKey` lookup results in a cache miss, a new CachedChunk instance will be created for that chunk. Additionally, the chunk buffer will need to be converted into the in-cache memory buffer format
    - If not, a cache hit occurs, indicating that the chunk already exists within the cache
- For each newly created `CachedChunk`, the dataset ID is checked, and if necessary[15], an instance of DsetList is created
    - For each chunk, the data buffer will be converted to the in-cache format with the appropriate buffer pointer being added to the `*cached_chunk_buf` field of the appropriate `CachedChunk` struct
    - After a chunk is added to a `DsetList`, the pointer to that dataset-focused structure is updated as the `*dset_ll_head`, with the other LL pointers being updated as necessary
    - The dataset configuration information is set and checked whenever a chunk is added or removed from the associated `DsetList`
- After all chunks have been added to the cache, I/O request(s) will be created to be as large as possible to minimize the amount of to-file write operations done[16]

3.3. Read To Cache From File

When reading raw data through the SCC, `H5SC_read()` is used. Similar to the write process, if datasets are supported by the SCC, they are processed in the following way[17]:

- The I/O request is passed into the SCC
- The size of the request is calculated
    - If the request will fit in the available memory, proceed without conducting eviction candidate selection
    - If not, proceed to eviction candidate selection
- For each chunk in the cache write request, the associated `ChunkKey` is computed

[15] If a dataset has existing `CachedChunk` elements in the SCC, an additional instance of `DsetList` will not be created.

[16] In the initial, minimal version of the SCC, I/O requests will be completed in smaller requests to ensure basic functionality of the cache.

[17] 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.

- The hash table is queried for each `ChunkKey` in the I/O request
    - If a `ChunkKey` lookup results in a cache miss, a new `CachedChunk` instance will be created for that chunk. Additionally, the chunk buffer will need to be converted into the in-cache memory buffer format
    - If not, a cache hit occurs, indicating that the chunk already exists within the cache
- For each newly created `CachedChunk`, the dataset ID is checked, and if necessary[18], an instance of `DsetList` is created
    - For each chunk, the data buffer will be converted to the in-cache format with the appropriate buffer pointer being added to the `*cached_chunk_buf` field of the appropriate `CachedChunk` struct
    - After a chunk is added to a DsetList, the pointer to that dataset-focused structure is updated as the `*dset_ll_head`, with the other LL pointers being updated as necessary
    - The dataset configuration information is set and checked whenever a chunk is added or removed from the associated `DsetList`
- After all chunks have been added to the cache, the I/O request(s) necessary for converting the requested chunks into the in-memory format and the scattering the buffers containing the chunk data into will be created to be as large as possible to minimize the amount of in-memory operations required to complete the request

3.4. 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 following procedure:

- Starting from the `*dset_ll_tail` pointer available in the `GlobalSCC` structure, each dataset is processed individually using `H5SC_flush_dset()`; when using this function, cached data may also be evicted (removed from the SCC) by setting the evict boolean to be True
- For each chunk in the associated `DsetList` structure[19]:

[18] If a dataset has existing `CachedChunk` elements in the SCC, an additional instance of `DsetList` will not be created.

[19] Partially written or read are omitted from this initial description, though decisions centered around this concept have been considered and will be discussed in the Detailed Discussion section of this document.

    - If the `dirty_flag` field is True, the chunk should be written to the file; the `chunk_counter` for that chunk is incremented to indicate that it has been examined and will need to be added to a write I/O requested[20]
    - If the `dirty_flag` is False and the evict Boolean is `True`, the `CachedChunk` is invalidated; the `CachedChunk` is removed from the associated `DsetList`, the `ChunkKey` is removed from the global hash table, and all memory associated with the `CachedChunk` entry is freed. If the evict Boolean is `False`, no further action is taken
- After all non-dirty chunks have been freed or processed, the necessary I/O request(s) are created to write the chunks to file
    - Once this process is completed, the `dirty_flag` is set to `False`, allowing for those chunks to be freed if the evict boolean was set to True in the `H5SC_flush_dset()` function
- If eviction is desirable and all chunks for a dataset have been processed (a `DsetList` contains no chunks), the appropriate `DsetList` structure is freed and removed from the underlying linked list
    - The `*dset_ll_tail` pointer is then updated
    - When `*dset_ll_tail` and `*dset_ll_head` are both NULL, no datasets are present within the cache, allowing for the other structures to be freed
    - By default, the `GlobalSCC` will not be freed by this process
- Once all the elements of the `DsetList` structure have been iterated through, the flush operation is marked as completed
    - Either all datasets have been cleared of dirty chunks, or all `CachedChunk` and their associated entries were removed from the SCC

3.5. 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:

- When `H5SC_destroy()` is invoked, starting from `*dset_ll_tail`, each dsetList is processed individually
    - All chunks, regardless of the status of the `dirty_flag` and `partial_io` flags, are removed from the associated dsetList, the `ChunkKey` is removed from the global hash table, and all memory associated with the `CachedChunk` entry is freed
- Once a `DsetList` contains no chunks, that structure is freed and removed from the underlying linked list
    - The `*dset_ll_tail` pointer is then updated

[20] Again, the end goal is to compose I/O requests in such a way that they can be handled in as few write operations as possible, but this will likely be reserved as a future optimization.

- When `*dset_ll_tail` and `*dset_ll_head` are both `NULL`, no datasets are present within the cache, the `GlobalSCC` is then freed
- If necessary, any configuration-based structures are also freed

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 hashable item, the key, payload (i.e. the values), and the hash handle may be maintained in a single structure. The proposed ChunkKey is well suited to serve as a key when using UTHash, and provides a unique, data-focused identification for each individual chunk[21]. The macros for UTHash are well documented (see citation 5) and could easily be modified if need be.

4.2. Eviction

Other

4.3. Public Configuration Structure

[21] 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.

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”

References

You may not have references, but if you do put them here. In general, it is good to include references to sources you used (see examples).

1. The HDF Group. “HDF5 Documentation,” http://www.hdfgroup.org/HDF5/doc/doc-info.html (June 14, 2010).
2. Karyampudi, V., and T. Carlson. 1988. Analysis and numerical simulations of the Saharan air layer and its effect on easterly wave disturbances. *J. Atmos. Sci.* 45: 3102-3136.
3. Martin, C. and Y. Zhang. 2005. The diverse functions of histone lysine methylation. *Nature Rev. Mol. Cell Biol.* 6, 838–849.
4. Shu, S., and L. Wu. 2009. Analysis of the influence of Saharan air layer on tropical cyclone intensity using AIRS/AQUA data, *Geophys. Res. Lett.*, *36*, L09809.
5. uthash: a hash table for C structures. (2025). Github.io. https://troydhanson.github.io/uthash/
6.

All authors should be included in reference lists unless there are more than five, in which case only the first author should be given, followed by 'et al.'.

Please follow the style below in the published edition of Nature in preparing reference lists.

* Authors should be listed by surname first, followed by a comma and initials of given names.

* Titles of all cited articles are required. Titles of articles cited in reference lists should be in upright, not italic text; the first word of the title is capitalized, the title written exactly as it appears in the work cited, ending with a full stop. Book titles are italic with all main words capitalized. Journal titles are italic and abbreviated according to common usage; authors can refer to Nature, the Index Medicus or the American Institute of Physics style manual for details.

* Volume numbers are bold. The publisher and city of publication are required for books cited. (Refer to published papers in Nature for details.)

* References to web-only journals should give authors, article title and journal name as above, followed by the URL in full – or the DOI if known – and the year of publication in parentheses.

* References to websites should provide author names if known (or the company name), the title of cited page, the full URL, and the year of posting, or date of page updated, in parentheses, if known.

Revision History

July 25, 2025:	Version 0 Shared with Elena for comment.
July 31, 2025:	Version 0.1 Shared with Elena for follow-up comments.