

# **RFC: Shared Chunk Cache Internal API**

Neil Fortner

John Mainzer

Vailin Choi

3/14/25

To support the implementation of sparse chunk storage in HDF5, and to improve the performance of raw data I/O in general in HDF5, we intent to implement a new chunk cache that is shared among all datasets in a file. While the initial implementation will only be used by structured chunks, this cache will also support standard chunked and contiguous layouts, and any future layout that breaks the dataset into (hyper)rectangular chunks and stores these chunks separately in the file. It could also be extended to work with external datasets in the future.

Unlike the current chunk cache, which is internal to the chunk layout, and operates at a low level, being managed by the chunk code, we envision the shared chunk cache operating at a high level, managing and coordinating all raw data I/O. This will allow us to implement a more uniform approach to dataset I/O, allow for better coordination in the multi dataset I/O case, and simplify the eventual conversion to concurrent thread-safety in the cache, especially when enforcing ordering of I/O operations from multiple threads. During I/O, the shared chunk cache will be invoked directly from `H5D__read()` and `H5D__write()`, immediately after generic I/O initialization routines.

To implement this, there will need to be two API layers for the shared chunk cache, a top level interface that the library calls into, and a layout callback interface for when the shared chunk cache needs to perform layout-specific operations.

This is a work in progress and will almost certainly change.

## **Data Formats**

Any given piece of raw data can exist in three different formats: the application memory buffer format, the chunk cache memory format, and the disk format. The application memory buffer format is well understood from the existing HDF5 API: simply an array of elements in the memory datatype, with variable length data stored as `hvl_t` elements containing pointers to the variable length arrays. The chunk cache memory format is determined by the layout, but is generally considered to be stored in a deserialized version of the file format, uncompressed but in the file datatype. For the sparse chunk layout, the chunk cache memory format will likely consist of a structure containing the `H5S_t` describing the defined values and pointers to buffers for the different sections. For the legacy chunk format, the chunk cache memory format will be identical to the on disk format with the exception of compression. Finally, the on disk format is a serialized version of the chunk memory format and is how

the chunk is stored in the file, with any compression, etc. applied. Data in the chunk cache will largely be held in the chunk cache memory format, with the exception that some chunks will be additionally held as buffers containing the on-disk format as chunks are preemptively encoded as they near the conditions for eviction from the cache. Writing data to one of these preemptively encoded chunks before eviction will then invalidate the file format buffer.

## Structures

While the details of the cache's structures are yet to be determined, we can make some broad statements about their general arrangement. The top level `H5SC_t` struct will contain general cache settings such as the preemption policy, memory limit, and others, as well as the current memory footprint, number of actual bytes used within that footprint, and an LRU (least recently used) list of chunks, that can be in any dataset, or possibly a different structure used for a different preemption method:

```
typedef struct H5SC_t {
    H5SC_preemption_policy_t preemption_policy;
    hsize_t memory_limit;
    <other options>;
    size_t nbytes_alloc;
    size_t nbytes_used;
    H5SC_chunk_t *LRU_head;
    H5SC_chunk_t *LRU_tail;
} H5SC_t;
```

Each dataset will also contain its own hash table (or possibly a different structure) used to index that dataset's cached chunks. This can take the form of a `UT_hash_handle` placed in the `H5D_shared_t` struct. The `H5SC_chunk_t` struct will need to contain the chunk buffer, the chunk's scaled coordinates, address, allocated size on disk, number of bytes allocated and used in memory, and whether it contains only information on the selected elements:

```
typedef struct H5SC_chunk_t {
    void *chunk;
    hsize_t scaled[H5S_MAX_RANK];
    haddr_t addr;
    hsize_t disk_size;
    size_t nbytes_alloc;
    size_t nbytes_used;
    bool contains_values;
} H5SC_chunk_t;
```

## Top Level API

These are the functions that are called by the upper layers of the dataset package in the HDF5 library and serve as the initial entry points to the H5SC (shared chunk cache) package.

```
H5SC_t *H5SC_create(H5F_t *file, H5P_genplist_t *fa_plist);
```

Creates a new, empty shared chunk cache. Will be called at file open time.

```
herr_t H5SC_destroy(H5SC_t *cache);
```

Destroys a shared chunk cache, freeing all data used. Does not flush chunks. Called at file close time.

```
herr_t H5SC_read(H5SC_t *cache, size_t count, H5D_dset_io_info_t *dset_info);
```

Reads raw data through the shared chunk cache. Called by H5D\_\_read() after initial generic setup. There may be datasets in the dset\_info array that do not support the shared chunk cache. These datasets must be ignored by the shared chunk cache. There may also be datasets that have skip\_io set. These datasets must also be skipped.

```
herr_t H5SC_write(H5SC_t *cache, size_t count, H5D_dset_io_info_t *dset_info);
```

Writes raw data through the shared chunk cache. Called by H5D\_\_write() after initial generic setup. There may be datasets in the dset\_info array that do not support the shared chunk cache. These datasets must be ignored by the shared chunk cache.

```
herr_t H5SC_flush(H5SC_t *cache);
```

Flushes all cached data.

```
herr_t H5SC_flush_dset(H5SC_t *cache, H5D_t *dset, bool evict);
```

Flushes all data cached for a single dataset. If evict is true, also evicts all cached data.

```
herr_t H5SC_direct_chunk_read(H5SC_t *cache, H5D_t *dset, hsize_t *offset, void *udata, void *buf, size_t *buf_size);
```

Reads the chunk that starts at coordinates give by offset directly from disk to buf, without any decoding or conversion. First flushes that chunk if it is dirty in the cache. If buf\_size is not large enough to read the entire chunk, nothing is read, and the needed size is returned in \*buf\_size.

```
herr_t H5SC_direct_chunk_write(H5SC_t *cache, H5D_t *dset, hsize_t *offset, void *udata, const void *buf);
```

Writes the chunk that starts at coordinates given by offset directly from buf to disk, without any encoding or conversion. First invalidates and evicts that chunk from cache if it is present.

```
H5S_t *H5SC_get_defined(H5SC_t *cache, H5D_t *dset, const H5S_t *file_space)
```

Returns a copy of `file_space` with only elements selected that are both selected in `file_space` and defined in `dset`. If `file_space` uses a point selection, the ordering of selected points will be preserved in the returned dataspace.

```
herr_t H5SC_erase(H5SC_t *cache, H5D_t *dset, const H5S_t *file_space)
```

Causes the elements selected in `file_space` to become undefined in `dset`. If `dset` does not support tracking defined elements, returns an error.

```
herr_t H5SC_set_extent_notify(H5SC_t *cache, H5D_t *dset, const hsize_t *old_dims);
```

Called after `H5Dset_extent()` has been called for a dataset, so the cache can recompute chunk indices, delete chunks, clear unused sections of chunks, etc.

## Contents of `H5D_dset_io_info_t`

The `H5SC_read()` and `H5SC_write()` functions take a pointer to an `H5D_dset_io_info_t` struct. This is an existing structure that contains information about a single dataset in an I/O operation. The shared chunk cache will not need all of the information contained in this struct, and some will be filled in by the cache code during I/O initialization. Here is the current definition of the struct:

```
typedef struct H5D_dset_io_info_t {
    H5D_t      *dset; /* Pointer to dataset being operated on */
    H5D_storage_t *store; /* Dataset storage info */
    H5D_layout_ops_t layout_ops; /* Dataset layout I/O operation function pointers */
    H5_flexible_const_ptr_t buf; /* Buffer pointer */

    H5D_io_ops_t io_ops; /* I/O operations for this dataset */

    H5O_layout_t *layout; /* Dataset layout information */
    hsize_t nelmts; /* Number of elements selected in file & memory dataspace */

    H5S_t *file_space; /* Pointer to the file dataspace */
    H5S_t *mem_space; /* Pointer to the memory dataspace */

    union {
        struct H5D_chunk_map_t *chunk_map; /* Chunk specific I/O info */
        H5D_piece_info_t *contig_piece_info; /* Piece info for contiguous dataset */
    } layout_io_info;

    const H5T_t *mem_type; /* memory datatype */
    H5D_type_info_t type_info;
    bool skip_io; /* Whether to skip I/O for this dataset */
} H5D_dset_io_info_t;
```

The cache code will need to initialize type conversion during I/O setup. During this initialization, it will need to fill in the `type_info` field, except for the `request_nlemts` field. Here is the current definition of `H5D_type_info_t`:

```
typedef struct H5D_type_info_t {
    /* Initial values */
    const H5T_t *mem_type; /* Pointer to memory datatype */
    const H5T_t *dset_type; /* Pointer to dataset datatype */
    const H5T_t *src_type; /* Pointer to source datatype */
    const H5T_t *dst_type; /* Pointer to destination datatype */
    H5T_path_t *tpath; /* Datatype conversion path */

    /* Computed/derived values */
    size_t src_type_size; /* Size of source type */
    size_t dst_type_size; /* Size of destination type */
    bool is_conv_noop; /* Whether the type conversion is a NOOP */
    bool is_xform_noop; /* Whether the data transform is a NOOP */
    const H5T_subset_info_t *cmpd_subset; /* Info related to the compound subset conversion functions */
    H5T_bkg_t need_bkg; /* Type of background buf needed */
    size_t request_nlemts; /* Requested strip mine */
} H5D_type_info_t;
```

Finally, there is some type conversion info that is global to the I/O instead of being specific to a single dataset, primarily the conversion buffers. These will need to be passed to the callbacks in a separate struct. The existing pathway places this information in the `H5D_io_info_t` struct, but for the shared chunk cache we will create a new struct focused only on global type conversion information. Here is the definition of this new struct that will be passed to layout callbacks that do type conversion:

```
typedef struct H5D_io_type_info_t {
    uint8_t *tconv_buf; /* Datatype conv buffer */
    bool tconv_buf_allocated; /* Whether the type conversion buffer was allocated */
    size_t tconv_buf_size; /* Size of type conversion buffer */
    uint8_t *bkg_buf; /* Background buffer */
    bool bkg_buf_allocated; /* Whether the background buffer was allocated */
    size_t bkg_buf_size; /* Size of background buffer */
    H5T_vlen_buf_info_t vlen_buf_info; /* Vlen data buffer and info */
    bool must_fill_bkg; /* Whether any datasets need a background buffer filled with destination contents */
    bool may_use_in_place_tconv; /* Whether datasets in this I/O could potentially use in-place type
                                conversion if the type sizes are compatible with it */
} H5D_io_type_info_t;
```

Currently we plan to have the shared chunk cache query MPI collective settings and the selection I/O setting, and track and report the actual MPI modes and actual selection I/O mode, so this info does not need to be passed in the shared chunk cache API.

# Layout Callbacks

These are the callback functions that individual layout types implement in order to enable the shared chunk cache to perform these operations in a layout-agnostic manner. Some of these functions accept an `hsize_t` \*parameter called `scaled`. This is an array containing the scaled coordinates of the chunk, where the coordinates are divided by the chunk dimensions so that adjacent chunks differ in their scaled coordinates by a value of one. In addition, in order for the shared chunk cache to calculate the logical locations of the chunks, the chunk dimensions will be promoted to the `H5D_shared_t` struct and made available for all layout types where it is valid (contiguous datasets will simply use the dataset dimensions). The layout will also need to specify whether it uses the legacy vlen/reference storage where data is stored outside the dataset, or whether the variable length data is stored in a separate section of the chunk.

```
typedef herr_t (*H5SC_chunk_lookup_t)(H5D_t *dset, size_t count, const hsize_t *scaled[] /*in*/, haddr_t *addr[] /*out*/, hsize_t *size[] /*out*/, hsize_t *defined_values_size[] /*out*/, size_t *size_hint[] /*out*/, size_t *defined_values_size_hint[] /*out*/, void **udata[] /*out*/);
```

Looks up count chunk address and size on disk. `defined_values_size` is the number of bytes to read if only the list of defined values is needed. `size_hint` is the suggested allocation size for the chunk (could be larger if the chunk might expand when decoded). `defined_values_size_hint` is the suggested allocation size if only the list of defined values is needed. If `*defined_values_size` is returned as 0, then all values are defined for the chunk. In this case, the chunk may still be decoded without reading from disk, by allocating a buffer of size `defined_valued_size_hint` and passing it to `H5SC_chunk_decode_t` with `*nbytes` set to 0. `*udata` can be set to anything and will be passed through to `H5SC_chunk_decode_t` and/or the selection or vector I/O routines, then freed with `free()` (we will create an `H5SC_free_udata_t` callback if necessary).

```
typedef herr_t (*H5SC_chunk_decode_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, bool partial_bound, void **chunk /*in,out*/, void *udata);
```

Decompresses/decodes the chunk from file format to memory cache format if necessary. Reallocs chunk buffer if necessary. On entry, `nbytes` is the number of bytes used in the chunk buffer. On exit, it shall be set to the total number of bytes used (not allocated) across all buffers for this chunk. On entry, `alloc_size` is the size of the chunk buffer. On exit, it shall be set to the total number of bytes allocated across all buffers for this chunk. Optional, if not present, chunk is the same in cache as on disk. `partial_bound` is true if the chunk was encoded with `partial_bound` set to true. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_decode_defined_values_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, bool partial_bound, void **chunk /*in,out*/, void *udata);
```

The same as `H5SC_chunk_decode_t` but only decodes the defined values. Optional, if not present, the entire chunk must always be decoded..

```
typedef herr_t (*H5SC_new_chunk_t)(H5D_t *dset, bool fill, size_t *nbytes /*out*/, size_t *buf_size /*out*/, void **chunk /*chunk*/, void **udata /*out*/);
```

Creates a new empty chunk. Does not insert into on disk chunk index. If `fill` is true, writes the fill value to the chunk (unless this is a sparse chunk). The number of bytes used is returned in `*nbytes` and the size of the chunk buffer is returned in `*buf_size`.

```
typedef herr_t (*H5SC_chunk_condense_t)(H5D_t *dset, size_t *nbytes /*in, out*/, void **chunk /*in, out*/, void **udata);
```

Reallocates buffers as necessary so the total allocated size of buffers for the chunk (`alloc_size`) is equal to the total number of bytes used (`nbytes`). Optional, if not present the chunk cache will be more likely to evict chunks if there is wasted space in the buffers.

```
typedef herr_t (*H5SC_chunk_encode_t)(H5D_t *dset, hsize_t *write_size /*out*/, hsize_t *write_buf_alloc /*out*/, bool partial_bound, const void *chunk, void **write_buf /*out*/, void *udata);
```

Compresses/encodes the chunk as necessary. If chunk is the same as `cache_buf`, leaves `*write_buf` as NULL. This function leaves chunk alone and allocates `write_buf` if necessary to hold compressed data, sets `*write_size` to the size of the data in `write_buf`, and sets `*write_size_alloc` to the size of `write_buf`, if it was allocated. `partial_bound` is true if the chunk is partially outside the bounds of the dataset. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_encode_in_place_t)(H5D_t *dset, size_t *write_size /*out*/, bool partial_bound, void **chunk /*in,out*/, void *udata);
```

The same as `H5SC_chunk_encode_t` but does not preserve chunk buffer, encoding is performed in-place. Must free all other data used.

```
typedef herr_t (*H5SC_chunk_evict_t)(H5D_t *dset, void *chunk, void *udata);
```

Frees chunk and all memory referenced by it. Optional, if not present `free()` is simply used.

```
typedef herr_t (*H5SC_chunk_insert_t)(H5D_t *dset, size_t count, const hsize_t *scaled[] /*in*/, haddr_t *addr[] /*in,out*/, hsize_t old_disk_size[], hsize_t new_disk_size[], void *udata[]);
```

Inserts (or reinserts) count chunks into the chunk index if necessary. Old address and size (if any) of the chunks on disk are passed as `addr` and `old_disk_size`, the new size is passed in as `new_disk_size`. This function resizes and reallocates on disk if necessary, returning the address of the chunks on disk in `*addr`.

```
typedef herr_t (*H5SC_chunk_selection_read_t)(H5D_t *dset, const H5S_t *file_space_in, bool partial_bound, void
*chunk /*in*/, H5S_t **file_space_out /*out*/, bool *select_possible /*out*/, bool *require_values /*out*/, void
*udata);
```

Called when the chunk cache wants to read data directly from the disk to the user buffer via selection I/O. If not possible due to compression, etc, returns `*select_possible=false`. Otherwise transforms the file space if necessary to describe the selection in the on disk format (returns transformed space in `file_space_out`). If no transformation is necessary, leaves `*file_space_out` as NULL. `chunk` may be passed as NULL, and may also be an in-cache chunk that only contains information on defined values. If `chunk` is passed as NULL and the callback requires a chunk to be passed with (at least) the defined values selection, this callback shall return `*require_values=true` and `*file_space_out=NULL`. Optional, if not present, chunk I/O is only performed on entire chunks or with vector I/O. The H5SC code checks for type conversion before calling this. `partial_bound` is true if the on-disk chunk was encoded with `partial_bound` set to true. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_vector_read_t)(H5D_t *dset, haddr_t addr, const H5S_t *file_space_in, bool partial_bound,
void *chunk /*in*/, size_t *vec_count /*out*/, haddr_t **offsets /*out*/, size_t **sizes /*out*/, bool *vector_possible
/*out*/, bool *require_values /*out*/, void *udata);
```

Called when the chunk cache wants to read data directly from the disk to the user buffer via vector I/O. If not possible due to compression, etc, returns `vector_possible=false`. Otherwise returns the vector of selected elements in offsets (within the file, not the chunk, this is why `addr` is passed in) and sizes, with the number of vectors returned in `vec_count`. `chunk` may be passed as NULL, and may also be an in-cache chunk that only contains information on defined values. If `chunk` is passed as NULL and the callback requires a chunk to be passed with (at least) the defined values selection, this callback shall return `*require_values=true` and `*file_space_out=NULL`. Optional, if not present, chunk I/O is only performed on entire chunks or with selection I/O. The H5SC code checks for type conversion before calling this. `partial_bound` is true if the on-disk chunk was encoded with `partial_bound` set to true. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_selection_write_t)(H5D_t *dset, const H5S_t *file_space_in, bool partial_bound, void
*chunk /*in*/, H5S_t *file_space_out /*out*/, bool *select_possible /*out*/, bool *require_values /*out*/, void *udata);
```

Called when the chunk cache wants to write data directly from the user buffer to the cache via selection I/O. If not possible due to compression, etc, returns `select_possible=false`. Otherwise transforms the file



space if necessary to describe the selection in the on disk format (returns transformed space in `file_space_out`). If no transformation is necessary, leaves `*file_space_out` as NULL. `chunk` may be passed as NULL, and may also be an in-cache chunk that only contains information on defined values. If `chunk` is passed as NULL and the callback requires a chunk to be passed with (at least) the defined values selection, this callback shall return `*require_values=true` and `*file_space_out=NULL`. Optional, if not present, chunk I/O is only performed on entire chunks or with vector I/O. The H5SC code checks for type conversion before calling this. `partial_bound` is true if the on-disk chunk was encoded with `partial_bound` set to true. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_vector_write_t)(H5D_t *dset, haddr_t addr, const H5S_t *file_space_in, bool partial_bound,
void *chunk /*in*/, size_t *vec_count /*out*/, haddr_t **offsets /*out*/, size_t **sizes /*out*/, bool *vector_possible
/*out*/, bool *require_values /*out*/, void *udata);
```

Called when the chunk cache wants to write data directly from the user buffer to the cache via vector I/O. If not possible due to compression, etc, returns `vector_possible=false`. Otherwise returns the vector of selected elements in offsets (within the file, not the chunk, this is why `addr` is passed in) and sizes, with the number of vectors returned in `vec_count`. `chunk` may be passed as NULL, and may also be an in-cache chunk that only contains information on defined values. If `chunk` is passed as NULL and the callback requires a chunk to be passed with (at least) the defined values selection, this callback shall return `*require_values=true` and `*file_space_out=NULL`. Optional, if not present, chunk I/O is only performed on entire chunks or with selection I/O. The H5SC code checks for type conversion before calling this.

```
typedef herr_t (*H5SC_chunk_scatter_mem_t)(H5D_dset_io_info_t *dset_info, H5D_io_type_info_t *io_type_info, const
H5S_t *mem_space, const H5S_t *file_space, const void *chunk, void *udata);
```

Scatters data from the chunk buffer into the memory buffer (in `dset_info`), performing type conversion if necessary. `file_space`'s extent matches the chunk dimensions and the selection is within the chunk. `mem_space`'s extent matches the entire memory buffer's and the selection within it is the selected values within the chunk, offset appropriately within the full extent. Optional, if not present, `chunk` is the same in memory as it is in cache, with the exception of type conversion (which will be handled by the H5SC layer). If the layout stores variable length data within the chunk this callback must be defined. `partial_bound` is true if the on-disk chunk was encoded with `partial_bound` set to true. If the dataset reported `partial_bound_chunks_different_encoding` as false, the setting of `partial_bound` is undefined.

```
typedef herr_t (*H5SC_chunk_gather_mem_t)(H5D_dset_io_info_t *dset_info, H5D_io_type_info_t *io_type_info, const
H5S_t *mem_space, const H5S_t *file_space, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, size_t
*alloc_size_total /*in,out*/, void *chunk, void *udata);
```

Gathers data from the memory buffer (in `dset_info`) into the chunk buffer, performing type conversion if necessary. `file_space`'s extent matches the chunk dimensions and the selection is within the chunk. `mem_space`'s extent matches the entire memory buffer's and the selection within it is the selected values within the chunk, offset appropriately within the full extent. Defines selected values in the chunk. `alloc_size_total` represents the total number of bytes allocated across all buffers for this chunk, and must be updated by this callback if that size changes. Optional, if not present, chunk is the same in memory as it is in cache, with the exception of type conversion (which will be handled by H5SC layer). If the layout stores variable length data within the chunk this callback must be defined.

```
typedef herr_t (*H5SC_chunk_fill_t)(H5D_dset_io_info_t *dset_info, H5D_io_type_info_t *io_type_info, H5S_t *space,
size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, size_t *alloc_size_total /*in,out*/, void *chunk, void *udata);
```

Propagates the fill value into the selected elements of the chunk buffer, performing type conversion if necessary. `space`'s extent matches the chunk dimensions and the selection is within the chunk. `alloc_size_total` represents the total number of bytes allocated across all buffers for this chunk, and must be updated by this callback if that size changes. Optional, if not present, chunk is the same in memory as it is in cache, with the exception of type conversion (which will be handled by H5SC layer). If the layout stores variable length data within the chunk this callback must be defined.

```
typedef herr_t (*H5SC_chunk_defined_values_t)(H5D_t *dset, const H5S_t *selection, void *chunk, void *udata, H5S_t
**defined_values /*out*/);
```

Queries the defined elements in the chunk. `selection` may be passed as `H5S_ALL`. These selections are within the logical chunk. Optional, if not present, all values are defined.

```
typedef herr_t (*H5SC_chunk_erase_values_t)(H5D_t *dset, const H5S_t *selection, size_t *nbytes /*in,out*/, size_t
*alloc_size /*in,out*/, void *chunk, void *udata, bool *delete_chunk /*out*/);
```

Erases the selected elements in the chunk, causing them to no longer be defined. If all values in the chunk are erased and the chunk should be deleted, sets `*delete_chunk` to true, causing the cache to delete the chunk from cache, free it in memory using `H5SC_chunk_evict_t`, and delete it on disk using `H5SC_chunk_delete_t`. These selections are within the logical chunk. Optional, if not present, the fill value will be written to the selection using `H5SC_chunk_fill_t`.

```
typedef herr_t (*H5SC_chunk_evict_values_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, void
*chunk, void *udata);
```

Frees the data values in the cached chunk and memory used by them (but does not reallocate - see `H5SC_chunk_condense_t`), but leaves the defined values intact. Optional, if not present the entire chunk will be evicted.

```
typedef herr_t (*H5SC_layout_query)(H5D_t *dset, hsize_t *chunk_dims, bool *encode_decode_necessary, bool
*partial_bound_chunks_different_encoding);
```

Queries data about the dataset from the layout client. The callback shall set the chunk dimensions in the `chunk_dims` array (the number of dimensions is the same as the rank of the dataset), whether encoding and decoding is necessary for chunks between cache and disk, and shall set whether chunks that are partially outside the bounds of the dataset are encoded differently (for example, they may not have filters applied). If `*partial_bound_chunks_different_encoding` is set to true, then chunks whose partial bound state changes will be re-encoded and re-inserted as necessary after the dataset extent changes to ensure they are encoded appropriately.

```
typedef herr_t (*H5SC_delete_chunk_t)(H5D_t *dset, const hsize_t *scaled /*in*/, haddr_t addr, hsize_t disk_size);
```

Removes the chunk from the index and deletes it on disk. Only called if a chunk goes out of scope due to `H5Dset_extent()` or if `H5SC_chunk_erase_values_t` returns `*delete_chunk == true`.

## Callback struct

The final callback structure for each layout class is therefore:

```
typedef H5SC_layout_ops_t {
    H5SC_chunk_lookup_t          lookup;
    H5SC_chunk_decode_t          decode;
    H5SC_chunk_decode_defined_values_t decode_defined_values;
    H5SC_new_chunk_t             new_chunk;
    H5SC_chunk_condense_t        condense;
    H5SC_chunk_encode_t          encode;
    H5SC_chunk_evict_t           evict;
    H5SC_chunk_encode_in_place_t encode_in_place;
    H5SC_chunk_insert_t          insert;
    H5SC_chunk_selection_read_t   selection_read;
    H5SC_chunk_vector_read_t      vector_read;
    H5SC_chunk_selection_write_t  selection_write;
    H5SC_chunk_vector_write_t     vector_write;
    H5SC_chunk_scatter_mem_t      scatter_mem;
    H5SC_chunk_gather_mem_t       gather_mem;
    H5SC_chunk_fill_t            fill;
    H5SC_chunk_defined_values_t   defined_values;
    H5SC_chunk_erase_values_t     erase_values;
    H5SC_chunk_evict_values_t     evict_values;
    H5SC_layout_query             layout_query;
    H5SC_delete_chunk_t          delete_chunk;
} H5SC_layout_ops_t;
```

# Code Flow Examples

## Raw Data Write

H5D\_\_write() will perform initial setup, then call H5SC\_write(). The chunk cache will, for each chunk, check if it is in cache, if not it will look up the chunk with H5SC\_chunk\_lookup\_t. If the lookup finds the chunk on disk, and the full chunk is not being overwritten, the cache will read the chunk from disk then decode it to chunk cache memory format with H5SC\_chunk\_decode\_t. Multiple chunks could be loaded at once using vector I/O. If the lookup does not find the chunk or it is being fully overwritten, a new chunk will be created with H5SC\_new\_chunk\_t, with fill set to false if it is being fully overwritten and true otherwise. The data will then be written to the in cache chunk using H5SC\_chunk\_gather\_mem\_t, performing any type conversion necessary.

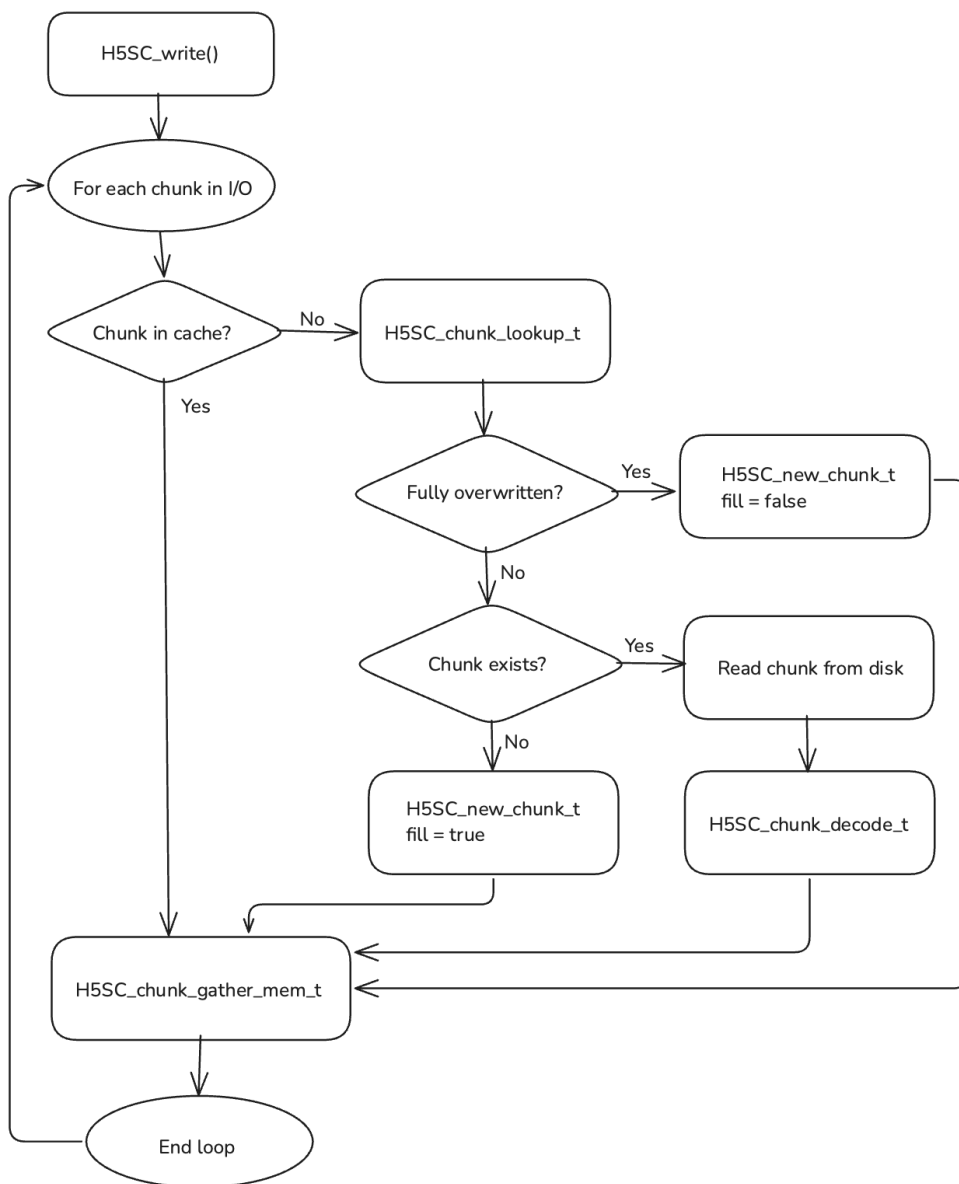


Figure 1: H5SC\_write()

As the cache starts to get full, it will intelligently assign some dirty chunks to preemptively encode the on-disk format (including compression if specified) using `H5SC_chunk_encode_t`. Once the cache is full, it will pick chunks to evict. For any such chunks that are dirty, it will, if `H5SC_chunk_encode_t` was called, evict the chunk with `H5SC_chunk_evict_t`, (re)insert the chunk into the index with `H5SC_chunk_insert_t`, write the data from the previously encoded write buffer to disk, then free the write buffer. If `H5SC_chunk_pre_flush_t` was not called, the cache will call `H5SC_encode_in_place_t`, (re)insert the chunk into the index with `H5SC_chunk_insert_t`, write the write buffer to disk, then free the write buffer. Clean chunks will simply be evicted with `H5SC_chunk_evict_t`.

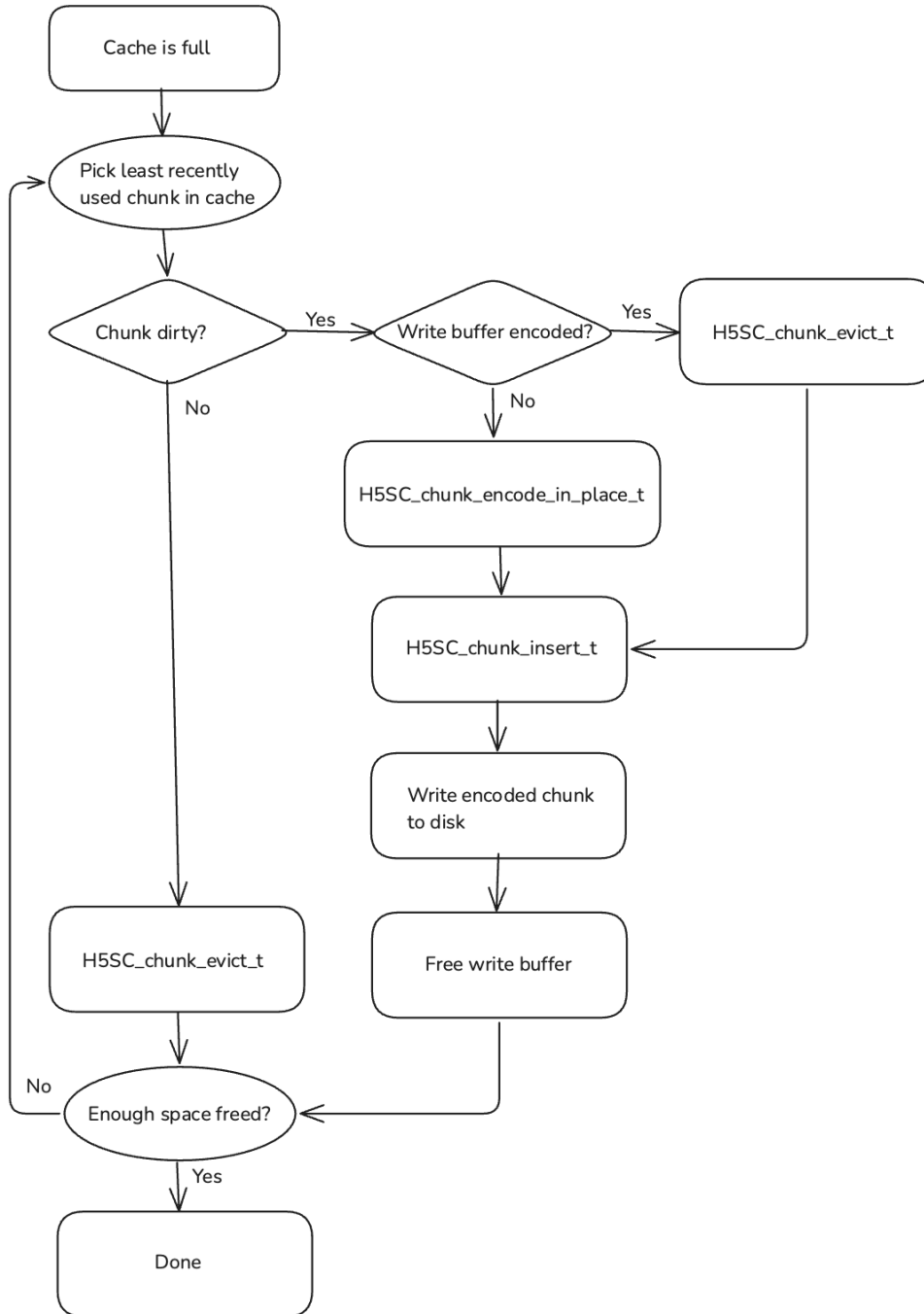


Figure 2: Chunk cache

eviction

## Raw Data Read

H5D\_\_read() will perform initial setup, then call H5SC\_read(). The chunk cache will, for each chunk, check if it is in cache, if not look up the chunk with H5SC\_chunk\_lookup\_t. If the lookup finds the chunk on disk, the cache will read the chunk from disk then decode it to chunk cache memory format with H5SC\_chunk\_decode\_t. Multiple chunks could be loaded at once using vector I/O. The data from the chunk in cache will be scattered to the memory buffer using H5SC\_chunk\_scatter\_mem\_t. If the lookup does not find the chunk, the cache will propagate the fill value to the selected elements in the memory buffer.

## Raw Data Write (Skip Cache)

If a raw data write operation will skip the cache for one or more chunks involved in I/O, either due to a user request, the chunk being too big, or if the cache decides it's best for some other reason, the shared chunk cache code will, for each dataset involved, check if H5SC\_chunk\_selection\_write\_t and/or H5SC\_chunk\_vector\_write\_t is defined. The cache will then iterate over chunks involved in the I/O that will skip the cache. For each chunk the cache will first look up the chunk's address with H5SC\_chunk\_lookup\_t. If the chunk does not exist on disk or neither H5SC\_chunk\_selection\_write\_t nor H5SC\_chunk\_vector\_write\_t are defined, the shared chunk cache will take the same actions as if the chunk cache were not being skipped except it will flush and evict the chunk immediately before moving on to the next chunk. Otherwise, if H5SC\_chunk\_selection\_write\_t is defined the shared chunk cache will invoke it with the correct file selection, and, if select\_possible is returned as true, issue a low level selection I/O request with the file selection returned and previously calculated memory selection (or add to a larger selection I/O op to issue later to cover all chunks or datasets). Otherwise, the shared chunk cache will similarly invoke H5SC\_chunk\_vector\_write\_t with the correct file selection, and, if vector\_possible is returned as true, calculate memory vectors to match the returned file vectors and issue a low level vector I/O call with these vectors (or add to a larger vector I/O op to issue later to cover all chunks or datasets).

If type conversion is required, the shared chunk cache will first check if the entire selection can fit in the type conversion buffer. If it can, it will proceed as above except the contents of the buffer will be gathered to the type conversion buffer, converted, and then this type conversion buffer will be passed as a contiguous source buffer to the low level selection or vector write routine (or add to a larger selection or vector I/O op if there is enough room in the type conversion buffer). If the selection cannot fit in the type conversion buffer, the shared chunk cache will only use the vector write callback, and if available it will process elements in batches up to the type conversion buffer size using a similar algorithm to the existing routine H5D\_\_scatgath\_write().

We could add a memory usage optimization to, if the chunk does not exist yet, only fill part of the chunk at a time. This will not be necessary to duplicate any existing behavior though, since contiguous datasets will allocate and fill the data iteratively using an existing separate pathway (H5D\_\_alloc\_storage) before the code reaches the shared chunk cache, and chunked datasets allocate and fill the entire chunk at once.

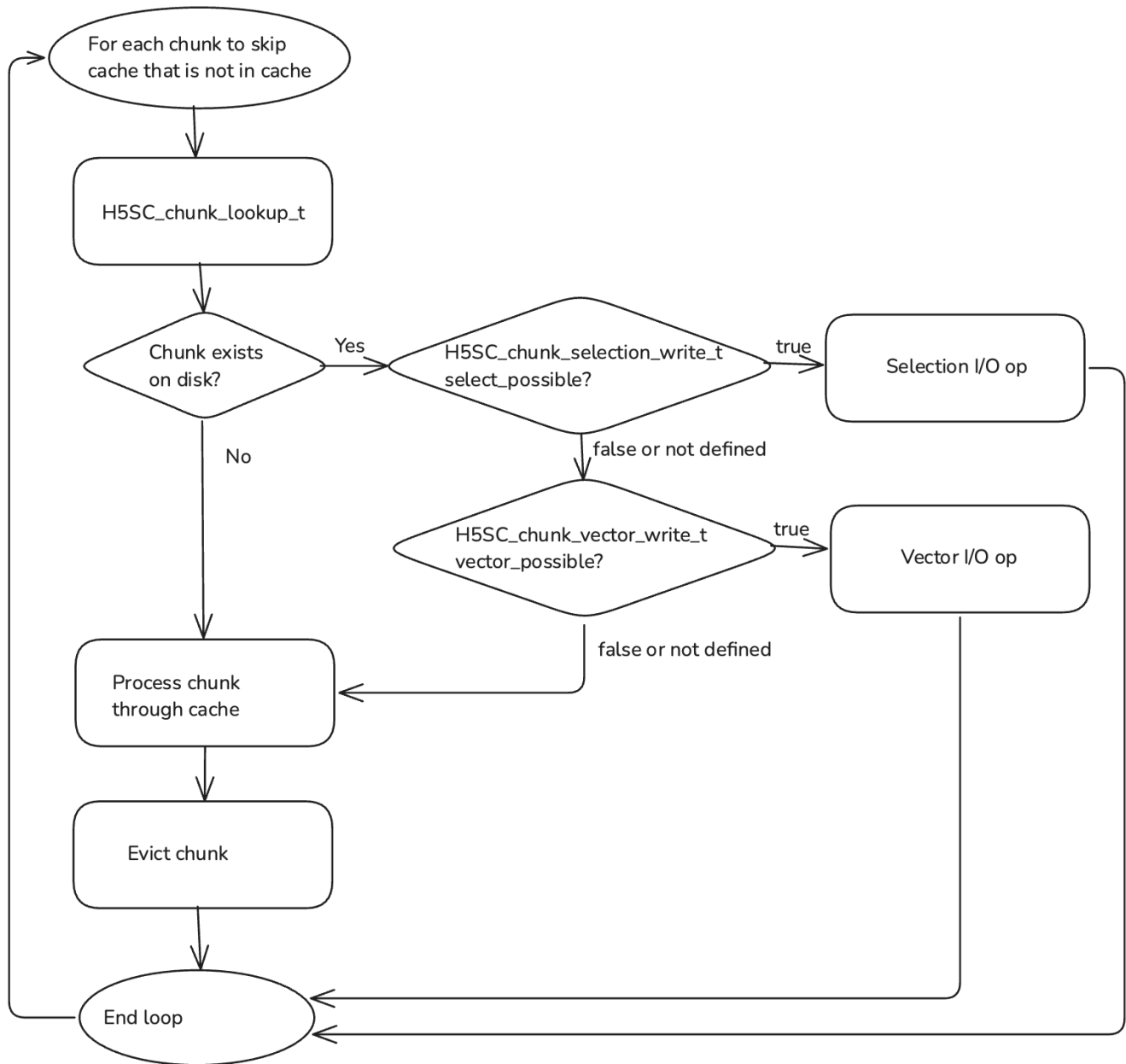


Figure 3: Raw data write bypassing cache

## Raw Data Read (Skip Cache)

If a raw data read operation will skip the cache for one or more chunks involved in I/O, either due to a user request, the chunk being too big, or if the cache decides it's best for some other reason, the shared chunk cache code will, for each dataset involved, check if `H5SC_chunk_selection_read_t` and/or `H5SC_chunk_vector_read_t` is defined. The cache will then iterate over chunks involved in the I/O that will skip the cache. For each chunk the cache will first look up the chunk's address with `H5SC_chunk_lookup_t`. If the chunk does not exist on disk the shared chunk cache will propagate the fill value to the matching selected parts of the user buffer. Otherwise, if `H5SC_chunk_selection_read_t` is defined the shared chunk cache will invoke it with the correct file selection, and, if `select_possible` is returned as true, issue a low level selection I/O request with the file selection returned and previously

calculated memory selection (or add to a larger selection I/O op to issue later to cover all chunks or datasets). Otherwise, if `H5SC_chunk_vector_read_t` is defined the shared chunk cache will similarly invoke it with the correct file selection, and, if `vector_possible` is returned as true, calculate memory vectors to match the returned file vectors and issue a low level vector I/O call with these vectors (or add to a larger vector I/O op to issue later to cover all chunks or datasets). Otherwise, the shared chunk cache will take the same actions as if the chunk cache were not being skipped except it will flush and evict the chunk immediately before moving on to the next chunk.

If type conversion is required, the shared chunk cache will first check if the entire selection can fit in the type conversion buffer. If it can, it will proceed as above except the type conversion buffer will be passed as a contiguous destination buffer to the low level selection or vector read routine (or add to a larger selection or vector I/O op if there is enough room in the type conversion buffer), then the type conversion buffer will be converted and the contents will be scattered to the application memory buffer. If the selection cannot fit in the type conversion buffer, the shared chunk cache will only use the vector read callback, and if available it will process elements in batches up to the type conversion buffer size using a similar algorithm to the existing routine `H5D__scatgath_read()`.

## **H5Dset\_extent()**

When performing a set extent operation, the upper levels of the library will first change the dataset struct, then make the `H5SC_set_extent_notify()` call. The shared chunk cache will then iterate over all of that dataset's chunks in cache, and recompute their chunk index. If the chunk has been completely removed from the extent, it will then be deleted on disk with the `H5SC_chunk_delete_t` callback and evicted from cache using the `H5SC_chunk_evict_t` callback and internal cache code to manage the cache structure. If the chunk is not completely removed from the extent but contains elements that are no longer in the extent, then, if `H5SC_chunk_erase_values_t` is defined, it will be called with the newly out of bounds elements as the selection. If `H5SC_chunk_erase_values_t` is not defined, then, if appropriate for the fill value/time settings, the fill value (or zero) will be written to the newly out of bounds elements using the `H5SC_chunk_scatter_mem_t` callback.

Next, the cache will iterate over all chunks that contain elements that are no longer in the extent (see `H5D__chunk_prune_by_extent()` for an example). If the chunk is in cache it is skipped, since it was already handled. Otherwise, if the chunk is still partly within the extent and (`H5SC_chunk_erase_values_t` is defined or (both `H5SC_chunk_selection_write_t` and `H5SC_chunk_vector_write_t` are not defined)), the chunk will be saved to a linked list of chunks to process (since the processing may cause chunks to be evicted from cache and subvert the selection criteria). It will then initiate a second pass over this linked list and, for each chunk, check if it exists on disk with `H5SC_chunk_lookup_t`. If it does not exist, processing can move to the next chunk, otherwise, it will be read and decoded using `H5SC_chunk_decode_t`. Next, if `H5SC_chunk_erase_values_t` is defined it will be invoked using the selection of newly out of bounds elements, otherwise the fill value will be written using `H5SC_chunk_fill_t`. If `H5SC_chunk_erase_values_t` is not defined and `H5SC_chunk_selection_write_t` or `H5SC_chunk_vector_write_t` is defined, only a single pass is necessary, and the fill value will be written directly to the newly out of bounds elements of each chunk using the procedure outlined above



for raw data write (skip cache), or if the chunk is no longer within the extent it will simply be deleted with `H5SC_chunk_delete_t`.

Finally, if early allocation is enabled, the cache will, if appropriate for the fill value settings, create a new chunk with the fill value using `H5SC_new_chunk_t` and encode it to on disk format using `H5SC_chunk_encode_in_place_t`. Next, the cache will iterate over all chunks that are newly within the extent, and allocate and insert each on disk using `H5SC_chunk_insert_t`, then, if the fill value chunk was created, write it to disk at the address returned by `H5SC_chunk_insert_t`. In the case of legacy vlen or reference types that store data elsewhere, the cache will need to avoid calling `H5SC_chunk_encode_in_place_t` at the start, and instead maintain a buffer of a memory type fill value chunk and convert and encode it anew for each chunk to be created, possibly using something like `H5D__fill_refill_vl()`.

## Early Allocation

Early allocation will be not be handled by the shared chunk cache, each layout type will implement it separately and no chunks will be cached by this operation.

## Appendix

### External Datasets

We would like to be able to extend the shared chunk cache to support external datasets. To do this, we recommend first modifying the external dataset code to use the `H5FD` layer to interact with external datasets. We may also want to add public API functions to allow the user to specify the file driver to use for the external data file. This change will cause an `H5FD_t *` to be stored within the `H5D_shared_t` struct (possibly in a nested struct), which can then be made visible to the shared chunk cache, either by placing the `H5FD_t *` in a uniform place or it can be returned through the low level API, possibly by adding an `H5FD_t **` to `H5SC_chunk_lookup_t` (non-external datasets would return `NULL`). The shared chunk cache will then proceed as normal, and whenever it needs to perform I/O to or from the disk it will simply use this `H5FD_t *` instead of the one associated with the dataset's (and cache's) file.

## Type Conversion

With the new sparse chunk format introducing a new way to store variable length and reference data types, we must reconfigure the internal datatype conversion interface to be able to handle this. To do this, we can add a new value to the `H5T_loc_t` enum so it looks something like:

```
typedef enum {
    H5T_LOC_BADLOC = 0, /* invalid datatype location */
    H5T_LOC_MEMORY, /* data stored in memory */
    H5T_LOC_DISK_VL_GHEAP, /* data stored on disk, with variable length data in a global heap */
    H5T_LOC_DISK_VL_INPLACE, /* data stored on disk, with variable length data co-located */
    H5T_LOC_MAXLOC /* highest value (Invalid as true value) */
} H5T_loc_t;
```

The new sparse chunk format will then use `H5T_LOC_DISK_VL_INPLACE`. At least initially, we will not allow conversion from disk to disk. Data conversion between `disk_vl_inplace` and memory will then involve an additional buffer, which will be used to store the variable length data for the data in disk (cache) format. This buffer will then need to be passed to `H5T_convert`, conversion callbacks, and made visible to the public API. To keep track of this buffer and information related to it, we can introduce a new struct:

```
typedef struct H5T_vlen_buf_info_t {
    void *buf;
    size_t nbytes_alloc;
    size_t nbytes_used;
} H5T_vlen_buf_info_t;
```

This will allow the type conversion code to reallocate the buffer if needed to fit more variable length data. Since the reallocation will only happen when converting from memory to cache format, the variable length data will already be in memory and excessive memory usage should not be a major issue. We may implement a first pass in the conversion step to determine the needed size of the vlen buffer. Here is the proposed signature for `H5T_convert()`:

```
herr_t H5T_convert(H5T_path_t *tpath, const H5T_t *src_type, const H5T_t *dst_type, size_t nelmts, size_t buf_stride,
size_t bkg_stride, void *buf, void *bkg, H5T_vlen_buf_info_t *vlen_buf_info);
```

The need to potentially reallocate the conversion buffer makes adding a public API function analogous to `H5Pset_buffer()` more complicated, but we could do so by passing the address of a buffer and an optional realloc callback.

## Encoded Format of Variable Length Data

Variable length and reference data in structured chunk datasets will be stored within the structured chunk, with the fixed length and variable length components stored in two separate sections. The fixed length section will contain a list of dataset elements of file datatype, where each element has the same constant size. Any variable length or reference datatypes in this file datatype will store, in this fixed length section, two 8 byte unsigned integers. The first integer represents the offset, in bytes, of the matching variable length data segment within the variable length section of the structure chunk, while the second integer represents the length, in bytes, of the variable length data segment. This variable length data segment can then be parsed according to standard HDF5 datatype rules - the number of elements of a variable length array can be inferred by the size of an element of the base type and the size of the data segment.

Nested variable length arrays will be flattened as follows: the top level is stored as above, with an offset/length (in bytes) pair stored in the fixed length section. In the variable length section, the start of a nested variable length sequence contains, before the sequence, an 8 byte integer representing the length, in elements of the base type, of this nested sequence, with the nested sequence stored

immediately after. Further levels of nesting follow this same pattern. In this way, an individual element of the top level datatype can be parsed with a single reading from the variable length section. This does mean the parser will not know, for nested variable length types, the number of elements of the highest level of variable length sequence, until parsing is complete.

## Email Correspondence

The following are emailed questions and accompanying answers that may help explain the interface.

Q: Why are `H5SC_chunk_lookup_t()` and `H5SC_chunk_decode_t()` callbacks separated? Isn't it the decoding is part of what `H5SC_chunk_lookup_t()` will do in order to find the chunk address and size on disk? My thoughts are that `H5SC_chunk_lookup_t()` will try to load the chunk index and decode it to find the chunk address and size on disk.

A: You are correct on what `H5SC_chunk_lookup_t` does. `H5SC_chunk_decode_t` will decode the chunk itself after it has been read from disk. This is necessary because the structured chunk code has a complicated format and I don't want the shared chunk cache to have knowledge of it. Also, standard chunked datasets may do decompression in this step. `H5SC_chunk_decode_t` decodes the chunk from the on disk format into the in cache format, which is also unknown to the shared chunk cache - it only knows the size and the buffer address.

Q: For `H5SC_chunk_lookup_t()` callback in the legacy chunked layout case, how do I figure out "size\_hint"? Is it the decompressed size for filtered chunk?

A: The size hint should probably be the maximum of the compressed and uncompressed sizes (the compressed size should be stored in the index, and is returned as size, and you can infer the uncompressed size with the datatype and chunk dimensions). If other info from the lookup will be needed it can be stored in `udata`. In the decode callback you will likely allocate a struct that has additional info and a pointer to the chunk buffer, and swap the "chunk" pointer to be this struct.

Q: For `H5SC_chunk_lookup_t()` callback in the legacy chunked layout case, will `*defined_values_size` and `*defined_values_size_hint` will be zero?

A: Yes.

Q: How does the structure look like for the chunk cache memory format that you mentioned on page 1 of the RFC?

A: I believe it will simply be the uncompressed, file datatype data chunk. However if something I haven't thought of comes up it may need an intermediate struct that contains a pointer to the raw data buffer and any additional info needed by the layout code.

Q: Regarding the `H5SC_chunk_decode()` and `H5SC_chunk_decode_defined_values_t()` layout callbacks for structured chunk, I don't quite understand why we need two separate callbacks? By the time they are called,

wouldn't the input parameters passed in be either for the whole chunk or for the defined values? In the `H5SC_chunk_lookup_t()` callback, I would need to decode the section for the encoded selection in order to obtain `defined_value_sizes` and `defined_values_size_hint` if this is not the whole chunk.

A: The lookup callback returns two separate sizes for the whole chunk and the defined values section. If the chunk cache only wants to read the defined values it can only read that size (which can be much smaller), and therefore the chunk buffer passed to `H5SC_chunk_decode_defined_values_t()` will only have enough information to decode the defined values, not the actual data. Passing this buffer to `H5SC_chunk_lookup_t()` would result in an error. For the legacy chunk case, `H5SC_chunk_decode_defined_values_t()` will be undefined/NULL. It would be possible to instead add a separate parameter to indicate if the data values should be decoded, but then we may need to add a separate method for the layout to report whether it supports decoding defined values. Also, if I'm reading the file format doc correctly, the sizes of the different sections are implicitly stored in the chunk index (as the section offsets), which is the the lookup callback's responsibility, so the lookup callback should not need to read the actual chunk.

Q: In the chunk-decode callback for structured chunk, I am going to decompress (if it's filtered) the [chunk] buffer passed in. To fill out the intermediate cache structure, I do need to do block read in order to retrieve and decode the `H5S_t` dataspace for the encoded selection, am I correct?

A: No, you should not need to do a block read. The encoded dataspace should be in the chunk buffer passed to the decode callback (along with everything else in the on-disk chunk).

Q: Ok, but I do need to do `H5S_decode` the encoded selection in the chunk buffer passed in to obtain the `H5S_t` dataspace?

A: Yes you will pass the appropriate portion of the chunk buffer to `H5S_decode`.

Q: For the intermediate chunk memory cache structure, should it be a field in `H5D_dset__io_info_t` or in `udata`?

A: The intermediate cache memory struct is not a field in `H5D_dset__io_info_t` or `udata`. It is filled in (and possibly reallocated) by the decode function, then the shared chunk cache will manage these structs in the cache until they are evicted.

Q: To double-check that I understand correctly:

For both the `chunk_decode` and the `chunk_decode_defined_values` callbacks, the parameter [chunk] will point to the whole structured chunk containing:

- section with the encoded selection
- section with data values

The `chunk_decode` callback will decode to fill in the intermediate structure with:

- `H5S_t` describing the selection
- buf pointing to the encoded selection
- buf pointing to the data values

The `chunk_decode_defined_values` callback will decode to fill in the intermediate structure with  
--H5S\_t describing the selection  
--buf pointing to the encoded selection

A: On entry, the chunk buffer pointer passed to `decode(_defined_values)` will point to a buffer that contains the raw byte stream on disk at the address returned by lookup, with the number of valid bytes equal to the size (or `defined_values_size`) returned by lookup. At the exit of `decode(_defined_values)`, the chunk buffer pointer should point to the intermediate struct that contains the information you need to implement the other callbacks such as gather and scatter. You probably don't need to keep the encoded selection around, just the H5S\_t. Though we could add that as an optimization later if use cases call for it (it would effectively be another layer of caching). Also, you'll need to add another buffer for the variable length data when that is implemented.

Q: Is the `H5SC_chunk_scatter_mem_t` layout callback similar to what is being done now in `H5D__scatgath_read()`?

A: It is somewhat similar but it does not read from the file. It occurs to me I might want to add a parameter to indicate if the callback can write to the chunk buffer, which would be set if the chunk is about to be evicted and would allow the callback to use the chunk buffer as the type conversion buffer. But this can be added later as an optimization.

Q: For `H5SC_chunk_scatter_mem_t` layout callback, are the input parameters `mem_space` and `file_space` the same as what is in the input parameter `dset_info`: `dset_info->mem_space`, `dset_info->file_space` ?

A: No, these parameters will be transformed to represent the selections within the chunk. The memory space's extent will be equal to the entire memory space, but the selection will only be the selected elements within the current chunk, and the file space's extent will be equal to the size of a single chunk.

Q: So these two input parameters are the already transformed selections within the chunk? So the callback will use those to scatter data from the "chunk" buffer to `dset_info->buf.vp`?

A: Yes that is correct, the shared chunk cache will compute the transformations.

Q: Also, the `dset_info->nelmts` and `dset_info->layout_io_info.contig_piece_info->in_place_tconv` are already setup for the chunk?

A: I don't think we'll set up `dset_info->nelmts`. If the shared chunk cache ends up needing to compute the number of elements selected in a chunk we can pass it a different way, otherwise the layout callback can compute it.

Q: For the `H5SC_chunk_encode_t` callback, is the input parameter `[chunk]` the pointer to the intermediate chunk memory cache structure?

A: Yes, it is the same as the `[chunk]` that was returned by `H5SC_chunk_decode_t`.

Q: For the H5SC\_chunk\_encode\_t callback, will the callback write the encoded selection plus the data to [write\_buf]?

A: Yes, it should place the encoded disk format buffer in write\_buf. This buffer will be written directly to disk.

Q: For the H5SC\_insert\_t callback, is the input parameter [chunk] the pointer to the intermediate chunk memory cache structure? Or is it the pointer to [write\_buf] obtained previously via H5SC\_chunk\_encode\_t callback?

A: Yes chunk is an array (length = "count") of pointers to cache format chunks (returned by decode\_t()), or NULL.

Q: For H5SC\_chunk\_insert\_t callback, does the count parameter mean there are [count] chunks to be inserted into the chunk index and they are consecutive chunks?

A: Yes, all the parameters with a "[]" are arrays of length count. They are not necessary "consecutive" in the dataset though.

Q: For the H5SC\_chunk\_encode\_t callback, if I need to do compression for the encoded selection and data, how do I get the filter masks for the two sections?

A: I believe the filter masks are calculated during the compression step, and stored in the file (I don't remember if they're stored in the index or the structured chunk header). All of this is done within H5SC\_chunk\_encode\_t. During decompression, they'll be read from the file and either placed in the udata by H5SC\_chunk\_lookup\_t (if stored in the index) or stored in the raw file format chunk (if stored in the header). Either way this data should be available to H5SC\_chunk\_decode\_t.

Q: For H5SC\_chunk\_decode\_t, I can obtain the filter mask for the sections from the udata passed in. But for H5SC\_chunk\_encode\_t, there's no input parameter udata?

A: The filter mask is calculated during the compression process within H5SC\_chunk\_encode\_t.

Q: I still have questions about the H5SC\_chunk\_insert\_t callback. I think the callback will:

If old\_disk\_size is different from new\_disk\_size, the callback will

--H5MF\_xfree(address/old\_disk\_size)

--H5MF\_alloc(new\_disk\_size) to get addr

--Insert (addr, new disk\_size) into the chunk index

I am not sure how the input parameter "chunk" is used in this callback?

A: This is the cache format chunk buffer itself. It's possible it won't be used by some clients, but I suspect it'll be needed in order to fill in the section offset fields for the index records for structured chunks. Though it occurs to me now, that we might want to add another udata to pass data between the encode and insert callbacks, analogous to the one used from lookup to decode. In this case the chunk probably won't be needed and could be removed from the signature.

Q: In the H5SC\_chunk\_insert\_t callback, do I have to do H5F\_shared\_block\_write(address, new or old disk\_size) to the file?

A: No, this is handled by the shared chunk cache. This is handled this way so the shared chunk cache can more easily accumulate multiple chunks into a single vector or selection I/O operation.

Q: I couldn't quite figure out the difference between the H5SC\_chunk\_evict\_t callback and the H5SC\_chunk\_evict\_values\_t callback. For the H5SC\_chunk\_evict\_t callback, the parameter [chunk] is the intermediate struct for the chunk's memory cache format that I defined as follows:

```
typedef struct H5D_chunk_mem_cache_t {  
    H5S_t select_ds; /* Dataspace for encoded selection */  
    void *buf; /* Buffer pointer to the data values */  
} H5D_chunk_mem_cache_t;
```

In H5SC\_chunk\_evict\_t callback, I will free the [buf], but how about H5SC\_chunk\_evict\_values\_t callback?

A: Evict\_values means only free the space used to store the actual values in cache, while leaving the struct intact (with the defined values selection). It occurs to me now that this sounds close to defined\_values which might be a problem, so maybe we can change the name.

In your example, evict\_values will free buf and set it to NULL. evict will free buf, close select\_ds, and free the chunk struct.

Q: As you indicated in the previous email that the [evict\_values] callback will just free buf and set it to NULL. I am puzzling on how do the input parameters [nbytes] and [alloc\_size] get used in the H5SC\_chunk\_evict\_values\_t callback?

A: alloc\_size refers to the total amount of memory space used (allocated) by the chunk (across all buffers), and nbytes refers to the actual amount of data used (specifically, the size that alloc\_size could be reduced to if the shared chunk cache directs the client to shrink its memory usage using H5SC\_chunk\_condense\_t. These fields allow the shared chunk cache to intelligently decide how to keep its memory footprint under the maximum while maximizing performance.

In this case, evict\_values will reduce \*alloc\_size by the (formerly) allocated size of buf, and reduce \*nbytes by the amount of space that was used in buf. You may need to add one or both of these numbers to the client chunk struct if they cannot be otherwise inferred.

Q: On exit from the `H5SC_new_chunk_t` callback, is the `[chunk]` a pointer to the intermediate chunk cache structure? How to determine the size of the new chunk? Is it the chunk size from `dset's` layout?

A: `H5SC_new_chunk_t` creates a new in-cache chunk populated with the fill value (if fill is set). The way this is encoded is up to the layout callback. For dense chunks it will probably be a full size array with the fill value propagated if appropriate, but you could add a flag to say if there are any defined values (essentially making it sparse-lite when in cache). For sparse chunks there will be no data buffer and the selection will be `NONE`. `buf_size` is, as before, the total amount of allocated memory in chunk, and `nbytes` is the size it could be compacted to if asked. They may be the same, unless you add as an optimization a larger initial allocation in the anticipation of the chunk growing.

Q: I don't quite understand the `H5SC_chunk_selection_read_t` callback, is it reading from the file to the `[chunk]`? Is it something like `H5D__select_read()` in `H5Dselect.c` that you are talking about? Is the parameter `[file_space_in]` the selection within the chunk? What do I need to do further to transform `[file_space_in]` to `[file_space_out]`?

Both the `chunk_selection_read/write` callbacks do not perform the actual read/write. The actual read/write are performed by the shared chunk cache code, right?

A: The purpose of `H5SC_chunk_selection_read_t` is for the layout to fill in the parameters that the shared chunk cache will pass to `H5F_shared_select_read()` (note `H5D__select_read()` is different).

`H5SC_chunk_selection_read_t` does no actual disk I/O, it simply sets up the operation. The reason it is done this way is so the shared chunk cache can aggregate selections from multiple chunks into a single call to `H5F_shared_select_read()`. This of course works similarly for write, as well as the vector routines.

`file_space_in` is the selection within the chunk. For legacy chunks you can simply leave `*file_space_out` as `NULL`, and the shared chunk cache will use `file_space_in` as `file_space_out`. For sparse chunks you may not want to define this callback, since it might be difficult/awkward to define the I/O pattern in terms of selections. I assume structured chunks will only use vectored I/O, at least initially. This does mean that there won't likely be a client that will initially return anything in `*file_space_out`, but I think it makes sense to leave it there for the sake of symmetry and as a reminder that the interface can be used this way. Clay, you're welcome to assert that `*file_space_out` is returned as `NULL` for now. It shouldn't be difficult to add in support for that later when it's needed.

Q: I don't need to define the `H5SC_chunk_fill_t` callback for structured chunk?

A: No, you will need to define that callback for structured chunk. It works similarly to `gather_mem`, except it only writes the fill value. You may not need to define it for standard chunk.

Q: `H5D__create()` will call "construct" callback to create layout information for the new dataset as below:

```
/* Create the layout information for the new dataset */
if (new_dset->shared->layout.ops->construct &&
    (new_dset->shared->layout.ops->construct)(file, new_dset) < 0)
```



```
HGOTO_ERROR(H5E_DATASET, H5E_CANTINIT, NULL, "unable to construct layout information");
```

I think similar ‘construct’ action needs to be done for structured chunk.

Does layout.ops defined/not defined for structured chunk? If not defined, how should “construct” action be done for structured chunk?

A: This operation does not relate to the cache, so it will bypass the shared chunk cache. You should add this callback to the existing layout\_ops, and implement it in a similar way to how the legacy chunk layout works.

Q: How do we implement erase\_values for structured chunk?

A: Here is an outline of how to implement the erase\_values callback:

1. Create a dataspace (H5S\_t) with rank 1 and extent equal to the number of elements selected in the chunk (before the erasure). We'll call this serial\_values\_space (you can of course use a different name in the code if you want). This dataspace represents the values as stored in the data section of the chunk.
2. Call H5S\_select\_project\_intersection with src\_space = the current defined\_values space for the chunk, dst\_space = serial\_values\_space, src\_intersect\_space = selection (the selection to erase, as passed to the callback), and share\_space = true. We'll call the output (returned in \*new\_space\_ptr) space serial\_erase\_space. This space represents a selection within the serialized value buffer of the elements to erase.
3. Iterate over the blocks of elements selected in serial\_erase\_space. There are a few different ways to do this, but the most common is probably get\_seq\_list. For each block of elements not erased memmove it down in the buffer to the end of the previous block of elements not erased, then update any data tracking the used space in the chunk and data block (including nbytes). Here's what this algorithm might look like:
  1. Set variables uint8\_t \*persist\_block\_start = 0, size\_t total\_erased\_bytes = 0
  2. Loop until all bytes processed:
  3. Obtain the start and end offset of the next block selected in serial\_erase\_space
  4. memmove the block from persist\_block\_start to (buf + (start \* dtype\_size)) down by total\_erased bytes (do nothing if total\_erased == 0)
  5. Increment total\_erased\_bytes by ((end - start) \* dtype\_size)
  6. Set persist\_block\_start = buf + (end \* dtype\_size)
  7. End loop
  8. If the last block of erase elements did not end at the end of serial\_erase\_space, memmove the block from persist\_block\_start to the end of the buffer down by total\_erased\_bytes.
  9. Reduce nbytes by total\_erased\_bytes.
  10. Assert that the correct number of bytes were erased.

This gets a bit more complicated when we add vlens and references. I'll keep in mind the needed functionality while implementing the new conversion routines and send an update to this algorithm in due time.

Q: I need some help on how to handle the following case in the `H5SC_chunk_gather_mem_t` callback for structured chunk:

--The application buffer in `[dset_info]` contains the data to be written to the chunk

--The `[chunk]` data buffer contains the existing chunk data read from disk and decoded via `H5SC_chunk_decode_t`

--After processing the data in the application buffer like type conversion, how do I combine/merge the data to be written with the existing chunk data read from disk? By iterating over `[mem_space]` and `[file_space]`?

A: Yes you'll need to iterate over the spaces. Look at `H5D__select_write()` (no type conversion) or `H5D__scatgath_write()` (type conversion) for an example on how to handle this. It may be worth creating versions of those functions for the callback to use so other places can use the same code. I'd be hesitant about trying to use `H5D__select_write()/H5D__scatgath_write()` as is though, since I'd like to try to avoid the weirdness with setting temporary layout operations etc.

Actually you may be able to use `H5D__select_io_mem()` as is when there's no type conversion. And possibly create an `H5D__scatgath_mem()` for when there is type conversion (this case is currently handled by setting layout ops to "compact" so the "file" ops still go to memory buffers).

Q: What is the parameter `[alloc_size_total]` in `H5SC_chunk_gather_mem_t` and `H5SC_chunk_gather_mem_t`?

A: That parameter represents the total amount of space used by the chunk buffer and any other buffers nested within it. This is to be used by the cache to track the memory usage of the chunk.

Q: How is the parameter `[chunk]` being used in `H5SC_chunk_vector_read_t` and `H5SC_chunk_vector_write_t` for structured chunk?

A: The chunk parameter contains the decoded chunk as held in the cache, the format of which is determined by the client. It's up to the client how to make use of it, but I imagine for structured chunk it would use the decoded dataspace selection to calculate the offsets of the requested values in the encoded/file format chunk. It probably doesn't make sense for the cache to call this callback if it has a complete chunk in cache, but it does if it caches only the defined values. Come to think of it, this is really required in the structured chunk case and we should probably include a way for the callback to request the chunk be loaded into cache with only the defined values. I'll try to update the API today. But I think implementing these callbacks should really be about the last thing to be done, since they're not required for correct behavior.