DRAFT RFC: Sparse Chunks

Neil Fortner

John Mainzer

8/30/18

1 Introduction

To date, the HDF5 library has supported only dense datasets – that is datasets in the form of n-dimensional matrices in which all entries are (typically) defined, either implicitly or explicitly.

However, there are potential HDF5 applications in which most entries in an n-dimensional matrix are explicitly undefined, and there is a need to both store the defined entries efficiently (in both space and time), and to identify and read defined entries efficiently as well. The following excerpt from an outline of the LCLS-II use cases should make this requirement clear:

> Assume a stream of large (1 – 4 mega pixel), 2 dimensional data sets (images if you wish) that are arriving at frequency f. The dimensions of the 2D datasets will not change over time.
>
> Further assume that for each 2D dataset, it is possible to automatically either:
>
> 1) Identify a rectangular Region Of Interest (ROI) in each 2D data set which will typically comprise about 10% of the 2 D dataset, and will change over time, or
>
> 2) Identify 50 – 100 small subsections in each 2D datasets (AKA a point list). The size of the subsections is typically small, and irregular – say 5 – 10 (typically) contiguous points or pixels. The number, size, configurations, and locations of the small subsections will change over time.
>
> For each 2D dataset in the stream, store only the ROI or the point list in a 3D dataset (the third dimension is the index of the 2D data set in the stream). Must be able to recover both the location and contents of the ROI or the elements of the point list as appropriate for each 2D dataset.
>
> In addition, for some n >= 1, store every nth 2D dataset in full. (n is constant over any given run. Typical values are in the range 1 – 10K). Do this either in the 3D dataset mentioned above, or in a separate 3D dataset[1] as convenient. Note that the ROI or point list of each 2D dataset that is stored in full must be recoverable as well.
>
> Overall objective is to severely reduce the quantity of data stored by discarding the "uninteresting" parts. Store every nth 2D dataset in full to permit verification of the correctness of the automatic recognition of ROIs or point lists as appropriate.

To meet this requirement, we must implement sparse datasets – that is datasets in which:

1 Or store the ROI or point list in the 3D dataset mentioned above, and the full 2D dataset in a second 3D dataset.

- Only the entries that have been written explicitly are defined.
- The defined entries can be readily identified, and read.

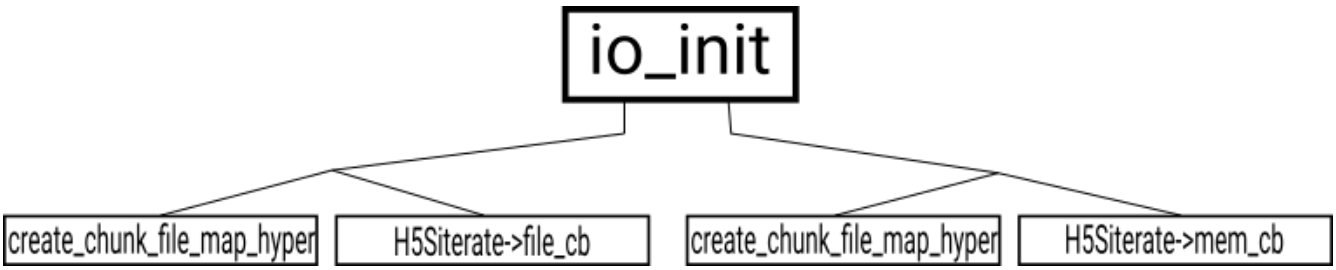
To the above minimal requirement, we also add:

- Compatibility with dense datasets – thus code designed for the existing dense datasets will still work, reading defined values if available, and the fill value (default 0) where not.
- Ability to erase defined values – that is to remove them from the set of defined values.

Several options for implementing sparse datasets were explored in the paper "Sparse Dataset Design Options". Of these, the Sparse Chunks approach seems to offer the best mix of match with the functional requirements as currently understood and ease of implementation. This RFC develops the Sparse Chunks design more fully, with an eye to addressing design details and estimating the cost of implementation.

2 Current Design

This section gives a brief overview of the most important sections of the chunk code involved in I/O, as they exist currently (prior to implementing sparse datasets). During an I/O operation, the dataset code first, after setting non-layout-specific information, calls the io_init layout callback, then either the read or write callback. The io_init call serves as a preliminary pass over the I/O operation, collecting information that is needed before the actual I/O begins. In the chunk code the primary purpose of H5D__chunk_io_init is create file and matching memory dataspaces for each chunkeach containing a selection of only the selected elements in that chunk. Included below are diagrams of the most important functions involved in each call and a description of each.



2.1 io_init

2.1.1 H5D__chunk_io_init

This function first performs general setup, then, if the selection is a hyperslab, calls H5D__create_chunk_file_map_hyper to generate file selections for each chunk, otherwise it calls H5S_select_iterate on the file space with H5D__chunk_file_cb as the callback function to do the same. Next, if the selection is a hyperslab and has the same "shape" in file and memory, it calls

H5D__create_chunk_mem_map_hyper to generate memory selections for each chunk, otherwise it initializes a selection iterator for the memory space and calls H5S_select_iterate on the file space with H5D__chunk_mem_cb as the callback function to do the same.

2.1.2 H5D__create_chunk_file_map_hyper

This function first creates a bounding box for the file selection, then iterates over all chunks contained in that box. For each chunk, it first checks to see if the file selection intersects with the chunk. If it does, it creates a dataspace representing the file chunk and containing the selected elements in the chunk, then adds the chunk to the skiplist of chunks involved in I/O.

2.1.3 H5D__chunk_file_cb

This function is called as a callback by H5S_select_iterate. H5S_select iterate makes this callback once for every element in the file selection. H5D__chunk_file_cb first identifies the chunk containing the selected element, then searches the skiplist to see if the chunk has been added yet. If it has not yet been added, the function creates a dataspace representing the chunk and adds it to the skiplist. In either case, it then adds the selected element to the chunk's file dataspace.

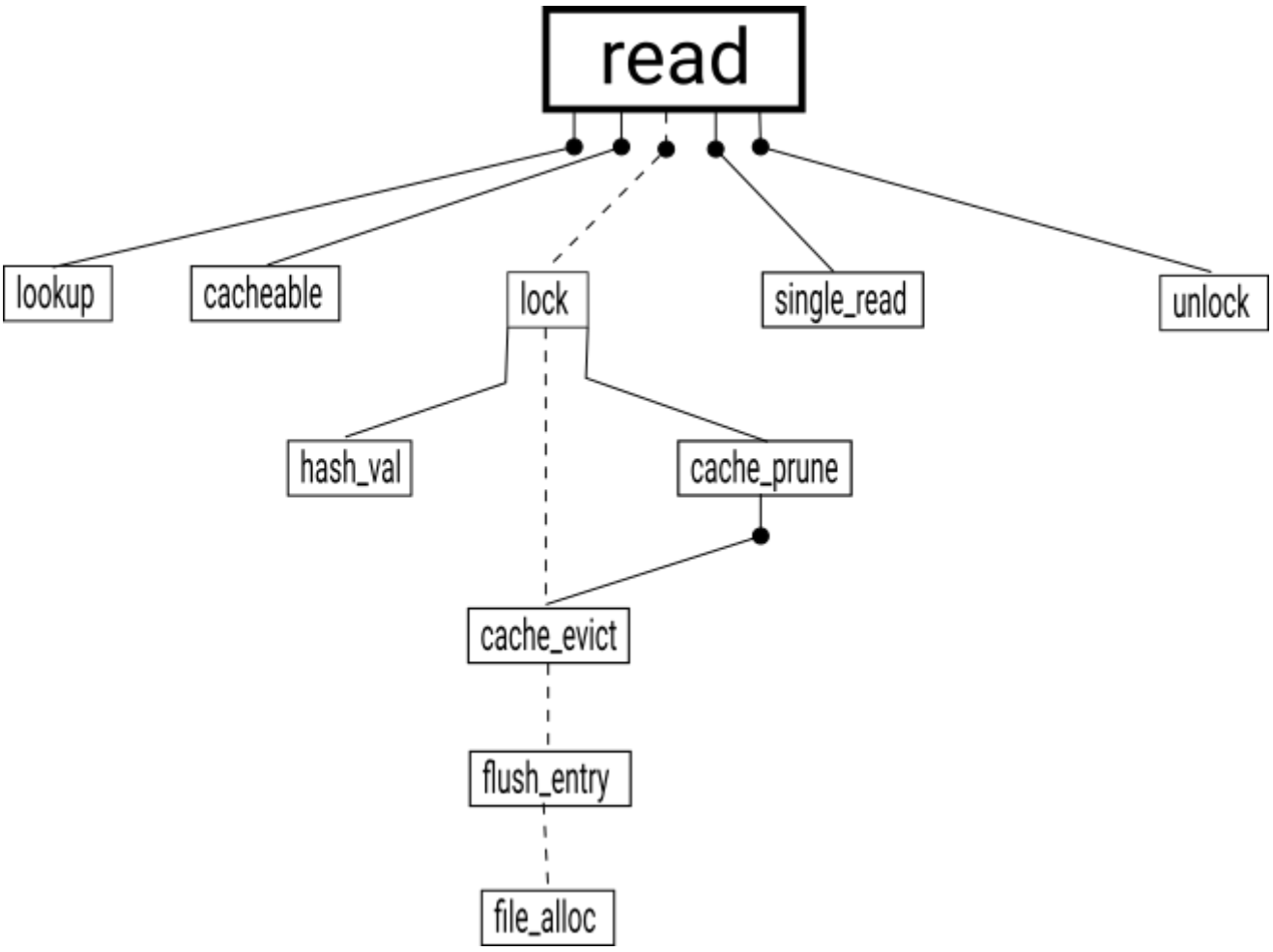
2.1.4 H5D__create_chunk_mem_map_hyper

This function iterates over all chunks identified by the file space iteration above in H5D__create_chunk_file_map_hyper, and for each chunk, creates a memory space and copies the selection from the file chunk space to the memory space, adjusting the offset as appropriate.

2.1.5 H5D__chunk_mem_cb

This function is called as a callback by H5S_select_iterate. H5S_select iterate makes this callback once for every element in the file selection. H5D__chunk_mem_cb first identifies the chunk containing the selected element, then obtains that chunk from the skiplist (it must have been added by H5D__chunk_file_cb), and creates a memory space for the chunk if it has not been created yet. Next, it obtains the next selected memory element from the iterator initialized by H5D__chunk_io_init and adds this element to the memory dataspace. Since the memory iterator is advanced at the same speed as the file iteration (once per callback), the memory and file selections will match correctly.

2.2 read



2.2.1 H5D__chunk_read

This function iterates over the chunks involved in I/O, as identified by H5D__chunk_io_init. For each such chunk, it first calls H5D__chunk_lookup to find the chunk in the file, then calls H5D__chunk_cacheable to determine if the chunk can be held in the chunk cache. If the chunk is cacheable, it then calls H5D__chunk_lock to obtain a pointer to the cached chunk (loading it into cache if necessary), and sets the chunk I/O to use the "compact" operations (I/O to/from memory buffer). Otherwise, it sets the chunk I/O to use the "contiguous" operations (I/O to from disk).

Next, H5D__chunk_read calls the "single read" routine, which for serial operations is either H5D__scatgath_read (in the case of type conversion) or H5D__select_read (otherwise). These functions iterate over the selections, perform type conversion in the first case, and call the chunk I/O ops set above. Finally, H5D__chunk_read calls H5D__chunk_unlock which releases the hold on the chunk, allowing it to be evicted when the chunk cache needs to make space.

2.2.2 H5D__chunk_lookup

The purpose of this function is to retrieve the address of the chunk on disk. It first checks if the chunk is in the cache. If it is, it returns the address from the cache and also returns the index of the chunk in the cache, signaling to the calling function that the chunk is cached. If not, it first checks if the chunk

was the last one looked up, since it keeps a separate cache of the address only for the last chunk. If it was not, it queries the chunk index for the chunk address, an operation which may result in metadata reads.

2.2.3 H5D__chunk_cacheable

This function checks to see if the chunk should be loaded into cache with H5D__chunk_lock. This function can return true even if the chunk will immediately be flushed out, since some cases are only handled by the code path that goes through H5D__chunk_lock. This function always returns true for filtered chunks and chunks that need to have the fill value written. Otherwise, it returns false for chunks that are too big to fit in cache, and whenever the file is opened in parallel with write access.

2.2.4 H5D__chunk_lock

This function loads a chunk into the chunk cache. First, it checks if the chunk is already in cache, and if so, moves the chunk one space towards the tail of the LRU list, locks the chunk in cache and returns. Otherwise, if it is not about to be fully overwritten (as signaled by the “relax” parameter), it reads the chunk from disk into the cache if it exists on disk, or allocates a chunk in memory and fills it with the fill value if it does not exist on disk.

Next, if the chunk can fit in the cache, it calls H5D__chunk_hash_val to get the index into the hash table, and checks if a chunk is already present at that index. If so, unless it is locked, the previous chunk with the same hash value is evicted from cache with H5D__chunk_cache_evict. After the hash table entry is clear, H5D__chunk_lock calls H5D__chunk_cache_prune to enough entries (if necessary) so the addition of the new chunk does not cause the chunk cache to exceed the maximum size. Next, the new chunk is added to the hash table, added to the LRU list at the tail, and locked in cache.

Note that there are various cases where H5D__chunk_lock can return a chunk that was not added to the cache, and will be freed when H5D__chunk_unlock is called.

2.2.5 H5D__chunk_hash_val

This function computes the hash value used as in index into the hash table used to index thechunks in cache. The hash value is computed by accumulating the scaled chunk coordinates (the coordinates divided by the chunk dimensions) into a single number using bitwise operations, then taking the modulus of that number by the hash table size.

2.2.6 H5D__chunk_cache_prune

This function makes space in the cache to hold a new chunk. If there is already enough space it returns immediately. Otherwise, it starts at the head of the LRU list, evicting chunks using H5D__chunk_cache_evict (with flush set to TRUE) that aren’t locked and have either been neither read from or written to, not written to but fully read from, or not read from but fully written to. After a certain percentage of the list has been traversed (this percentage is controlled by the “w0” parameter in H5Pset_cache and H5Pset_chunk_cache) it begins simultaneously traversing the list from the head and evicting all unlocked chunks (while the stricter eviction method continues from the point it left off). LRU list traversal ends as soon as there is enough space to hold the chunk.

2.2.7 H5D__chunk_cache_evict

This function evicts a chunk from the cache. If the flush parameter is set to true, it calls H5D__chunk_flush_entry with reset set to TRUE, otherwise it simply frees the chunk buffer. In either case, it then removes the chunk from the LRU list and the hash table, and frees the entry.

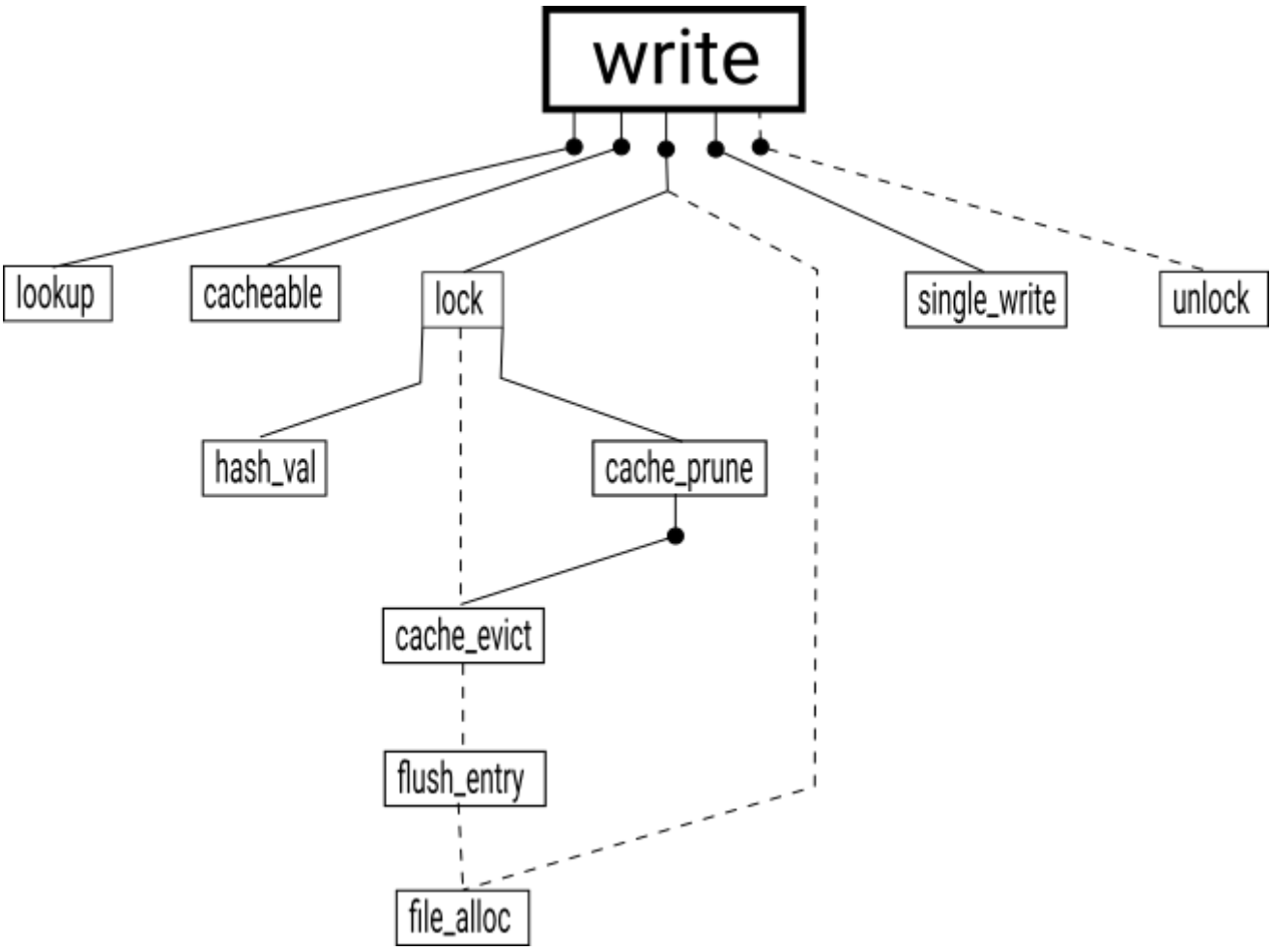
2.2.8 H5D__chunk_flush_entry

This function flushes a chunk to disk. If the chunk is dirty, it first runs the chunk through the filter pipeline if present, then, if it was filtered or doesn't exist on disk yet, calls H5D__chunk_file_alloc. Next, it writes the chunk to disk with H5F_block_write and, if need_insert was set by H5D__chunk_file_alloc, inserts the chunk into the on-disk chunk index using the indexing method's insert callback. Finally, whether or not the chunk was written, if reset is TRUE, it frees the chunk data buffer.

2.2.9 H5D__chunk_file_alloc

This function allocates space on disk for the new chunk. It first frees disk space used by the previous version of the chunk (if any), then allocates new space using H5MF_alloc (or calls the "get_addr" callback in the case of the "none" index).

2.3 write



2.3.1 H5D__chunk_write

This function iterates over the chunks involved in I/O, as identified by H5D__chunk_io_init. For each such chunk, it first calls H5D__chunk_lookup to find the chunk in the file, then calls H5D__chunk_cacheable to determine if the chunk can be held in the chunk cache. If the chunk is cacheable, it then calls H5D__chunk_lock (with reset set to TRUE if the entire chunk is to be written) to obtain a pointer to the cached chunk (loading it into cache if necessary), and sets the chunk I/O to use the “compact” operations (I/O to/from memory buffer). Otherwise, it calls sets the chunk I/O to use the “contiguous” operations (I/O to from disk) and, if the chunk hasn’t been allocated on disk, it calls H5D__chunk_file_alloc.

Next, H5D__chunk_write calls the “single read” routine, which for serial operations is either H5D__scatgath_write (in the case of type conversion) or H5D__select_write (otherwise). These functions iterate over the selections, perform type conversion in the first case, and call the chunk I/O ops set above. Finally, if the chunk was locked, H5D__chunk_write calls H5D__chunk_unlock which releases the hold on the chunk, allowing it to be evicted when the chunk cache needs to make space. If the chunk was not locked, H5D__chunk_write calls the index method’s insert callback to insert the new chunk into the on-disk index.

To do: Parallel I/O, allocation, set_extent, object copy

3 Sparse Chunks Approach to Sparse Datasets

3.1 Conceptual Overview

The basic idea of the Sparse Chunks approach is to use the existing HDF5 selection mechanism[2] to represent sparse datasets, both in memory and on disk. As it will in general be impractical to hold entire sparse datasets in memory, it will be convenient to break the extent of the sparse datasets into user specified, regular, n-dimensional rectangles. For each such rectangle, create a selection of all defined entries that lie within the target rectangle and call it a sparse chunk. Observe that this allows us to operate on one sparse chunk at a time.

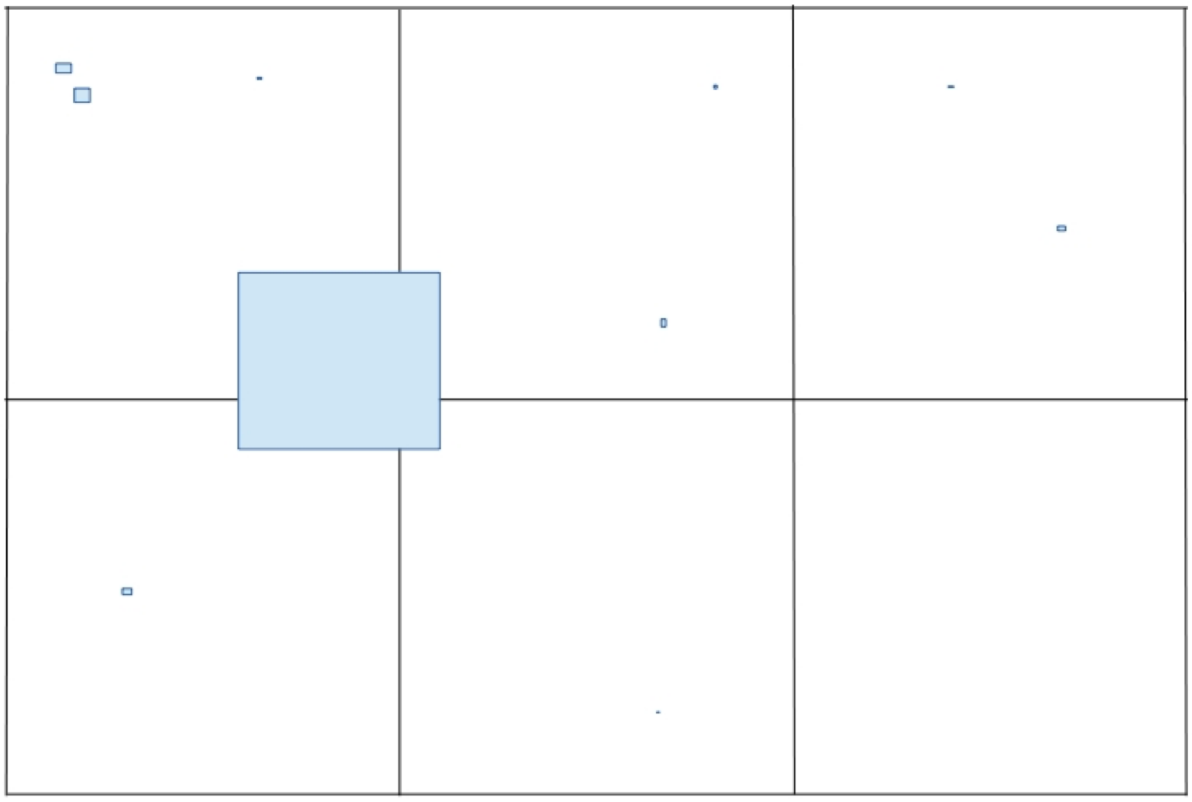
The existing facilities to:

- Serialize,
- De-serialize, and
- Compute intersections, unions, set subtractions

on selections give us the basic operations needed to support this. Disk I/O would be handled via modifications to the chunk cache to support store and load of region selections as required.

2 At present, selections in HDF5 must be either point or hyperslab – there is no facility for combining them. As hyperslab selections have no problem with hyperslabs consisting of a single point, there is no functional limitation here, although there are convenience issues.

Illustration



1: A conceptual representation of a 2 D sparse dataset in "sparse chunks" format. Each of the six squares represents a sparse chunk. The blue rectangles are members of the selection of defined points

As each sparse chunk will in general have a different on disk footprint, some variation of the existing mechanism for filtered chunks is needed to allocate file space. While in principle, this can be managed in the parallel case using much the same mechanism as used to implement parallel compression; the initial implementation will almost certainly be serial only.

An obvious objection to this design is the problem of storing semi-sparse data – if, for a given sparse chunk, the majority of entries are defined, the selection for that sparse chunk will likely be larger than the equivalent chunk in an equivalent dense dataset. We argue that this is a minor issue for the following reasons:

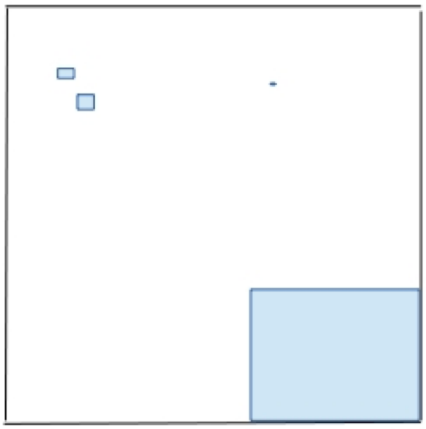
- The closest we come to the semi-sparse case in known use cases is the ROI case discussed in the introduction. In this case, the defined entry selection for the entire dataset is a single hyperslab – which imposes very little overhead even if an entire sparse chunk happens to lie in the ROI.

- It should be possible to optimize defined entry selections – i.e. to combine adjacent selections into larger hyperslabs where possible, to convert irregular hyperslab selections into regular

selections, etc. This is not a panacea, and should not be considered for the initial implementation. It is, however, a way of mitigating the issue if required.

In a nut-shell, this issue appears to be a hypothetical (for the EOD case) – and we have a strategy for mitigation if required. It should also be noted that this is a fundamental issue with sparse data representations – in the applied math domain, the usual solution is to move to a dense representation when the sparse representation becomes cost ineffective. Given that the motivation for sparse datasets in the EOD domain is data reduction, the same logic should apply.

Illustration



2: The upper left hand sparse chunk from the above illustration. Observe that it only contains that

4 New API Routines

4.1 H5P_SET_SPARSE_CHUNK

4.1.1 Signature

```
hid_t H5Pset_sparse_chunk(hid_t plist, int ndims, const hsize_t *dim)
```

4.1.2 Parameters

`plist`	IN: Dataset creation property list identifier
`ndims`	IN: The number of dimensions of each chunk
`dim`	IN: An array defining the size, in dataset elements, of each chunk

4.1.3 Description

`H5P_SET_CHUNK` sets the size of the chunks used to store a sparse chunked layout dataset. This function is only valid for dataset creation property lists.

The `ndims` parameter currently must be the same size as the rank of the dataset.

The values of the dim array define the size of the chunks to store the dataset's raw data. The unit of measure for dim values is *dataset elements*.

As a side-effect of this function, the layout of the dataset is changed to H5D_SPARSE_CHUNKED, if it is not already so set. (See H5P_SET_LAYOUT.)

The sparse chunk layout only stores elements that are defined, and keeps track of which elements are defined. Initially, all elements are not defined, and writing to an element causes it to be defined. A selection containing defined elements can be obtained using H5P_GET_DEFINED. Defined elements can be removed from the dataset using H5D_ERASE.

- Chunk size cannot exceed the size of a fixed-size dataset. For example, a dataset consisting of a 5x4 fixed-size array cannot be defined with 10x10 chunks.
- Chunk maximums
    - The maximum number of elements in a chunk is 2^{32}-1 which is equal to 4,294,967,295. If the number of elements in a chunk is set via H5P_SET_CHUNK to a value greater than 2^{32}-1, then H5P_SET_SPARSE_CHUNK will fail.
    - The maximum size for any chunk is 4GB. If a chunk that is larger than 4GB attempts to be written with H5D_WRITE, then H5D_WRITE will fail.

4.1.4 Returns

Returns a non-negative value if successful; otherwise returns a negative value.

4.2 H5P_GET_SPARSE_CHUNK

4.2.1 Signature

```
int H5Pget_sparse_chunk(hid_t plist, int max_ndims, hsize_t *dims)
```

4.2.2 Parameters

plist	IN: Identifier of property list to query
max_ndims	IN: Size of the dims array
dims	OUT: Array to store the chunk dimensions

4.2.3 Description

H5P_GET_SPARSE_CHUNK retrieves the size of chunks for the raw data of a sparse chunked layout dataset. This function is only valid for dataset creation property lists. At most, max_ndims elements of dims will be initialized.

4.2.4 Returns

Returns chunk dimensionality if successful; otherwise returns a negative value.

4.3 H5D_GET_DEFINED

4.3.1 Signature

```
hid_t H5Dget_defined(hid_t dataset_id, hid_t file_space_id, hid_t xfer_plist_id)
```

4.3.2 Parameters

dataset_id	IN: Identifier of the dataset to get the selection of defined elements from
file_space_id	IN: Identifier of the selection in the file dataspace of elements to be queried if they are defined, or H5S_ALL if all defined elements in the dataset are desired
xfer_plist_id	IN: Identifier of a transfer property list for this I/O operation

4.3.3 Description

H5D_GET_DEFINED retrieves a dataspace object with only the defined elements of a (subset of) a dataset selected. The dataset is specified by its identifier dataset_id, and data transfer properties are defined by the argument xfer_plist_id. The subset of the dataset to search for defined values is given by the selection in file_space_id. Setting file_space_id to H5S_ALL causes this function to return a selection containing all defined values in the dataset.

This function is only useful for datasets with layout H5D_SPARSE_CHUNKED. For other layouts this function will simply return a copy of file_space_id, as all elements are defined for non-sparse datasets.

4.3.4 Returns

Returns a dataspace with a selection containing all defined elements that are also selected in file_space_id if successful; otherwise returns a negative value.

4.4 H5D_ERASE

4.4.1 Signature

```
herr_t H5Derase(hid_t dataset_id, hid_t file_space_id, hid_t xfer_plist_id)
```

4.4.2 Parameters

dataset_id	IN: Identifier of the dataset to erase elements from
file_space_id	IN: Identifier of the selection in the file dataspace of elements to be erased
xfer_plist_id	IN: Identifier of a transfer property list for this I/O operation

4.4.3 Description

H5D_ERASE erases elements from a dataset, specified by its identifier dataset_id, causing them to no longer be defined. After this operation, reading from these elements will return fill values, and the elements will no longer be included in the selection returned by H5D_GET_DEFINED. Data transfer properties are defined by the argument xfer_plist_id. The part of the dataset to erase is defined by file_space_id.

This function is only useful for datasets with layout `H5D_SPARSE_CHUNKED`. For other layouts this function will return an error.

4.4.4 Returns

Returns a non-negative value if successful; otherwise returns a negative value.

4.5 Rationale

We believe these four functions will be sufficient to cover most uses cases with good performance. Consider the use case where an application wants to determine which elements are defined and read only those elements. It might be tempting to create an API routine to perform both of these operations in one call, and it might seem at first that using this routine would reduce the amount of file I/O. However, consider that, if working one chunk at a time, this routine will need to know the total size of the memory buffer to avoid reallocation, and it will need to know the shape of the entire file selection (the defined elements) before it can determine where in the memory buffer the defined elements should be placed. Therefore, this routine must either make two passes, first reading each chunk for its defined element selection then reading each chunk's data, or it must hold all chunks in memory until the operation is complete.

Both of these methods can be functionally accomplished using only H5D_GET_DEFINED and existing routines. In both cases the application calls H5D_GET_DEFINED to get the selection of defined elements, then calls H5D_READ with that selection. If the chunk cache is set to a small size, it will be equivalent to the two pass approach, while if the chunk cache is set large enough to hold all the selected chunks, then it will be equivalent to the latter approach. It may be helpful in the future to add routines to change the size of the chunk cache without reopening the dataset.

5 Approach

Implementing the sparse chunks scheme described above presents unique challenges. Since it shares many similarities with the existing chunks implementation, it makes sense to use some of that code. However, they way to do so is not clear. Integrating support for sparse chunks into the existing code risks making the code difficult to read, understand and maintain, due to special cases being inserted into existing algorithms (this is already a problem at present and would be made worse). Creating a separate source file for sparse chunks and copying needed code adds maintainability issues associated with code duplication. Finally, creating a separate source file and spinning off shared functionality into common functions would require substantial effort and could negatively impact readability due to strange or surprising delimitation of functionality. An alternative approach is described below.

6 Proposal: Chunk Format Layer

In order to facilitate implementing sparse chunks while using the existing chunk mechanisms, and to hopefully reduce the complexity of the current chunk code, this document proposes creating a new internal pluggable layer, called the chunk format layer, for operations on a single chunk. All I/O requests will go through this layer, and the upper level chunking code will have no knowledge of the

format of individual chunks, either in memory (cache) or on the disk, other than knowledge of whether the chunks on disk have variable size.

The upper level chunk code will, instead of dealing directly with the data, make calls to the chunk format layer interface to manipulate chunks. The chunk code will instruct the format layer to translate between memory and file chunk formats, transfer data between the memory buffer and a memory format chunk, and perform I/O directly to/from the file. The chunk format layer will operate only on individual chunks, and need not be aware of the full scope of the I/O operation.

File format chunks are simply an image of the chunk as it exists in the file. This allows the generic chunk code to perform file I/O on whole chunks. Partial chunk I/O that skips the cache must be performed by the chunk format layer, since the generic chunk code cannot “look into” the chunk and see the individual elements.

Memory format chunks are in a format, defined by the chunk format layer, that is convenient to manipulate while in memory. It should, for example, be uncompressed, and may have structured metadata (such as a dataspace) stored in structures and referenced by pointers, instead of in serialized form. Chunks held in the chunk cache will be in this form.

With this scheme, the current chunk implementation would be split into two formats: filtered and unfiltered chunks. The sparse chunks implementation will consist of an additional one or two formats: either a single format for all sparse chunks or one for filtered and one for unfiltered sparse chunks.

6.1 Chunk Format Struct Variables

6.1.1 variable_size

```
hbool_t variable_size;
```

This variable is a boolean type which indicates whether chunks on disk have a variable size. This is an important distinction that affects how chunks are allocated and whether they need to be reallocated on write.

6.2 Chunk Format Struct Functions

6.2.1 init

```
typedef void *(*H5D_chunk_format_init_t)(...);
```

This function is called when a dataset is first opened, and allows the format to save any data it needs to use for subsequent calls, and set up its internal structure. Exact parameters to be determined.

6.2.2 term

```
typedef herr_t (*H5D_chunk_format_term_t)(void *chunk_format_info);
```

This function is called when the dataset is shutting down, and it needs to close the chunk format. It should free all memory used by the format.

6.2.3 file_alloc

```
typedef void *(*H5D_chunk_format_file_alloc_t)(void *chunk_format_info, hsize_t
size);
```

This function should allocate space in memory for a file format chunk (i.e. an image of the chunk as it exists in the file).

6.2.4 file_to_mem

```
typedef void *(*H5D_chunk_format_file_to_mem_t)(void *chunk_format_info, const void
*file_chunk, hsize_t file_chunk_size, size_t *mem_chunk_size, hbool_t
disable_filters);
```

This function translates a file format chunk into a memory format chunk. It takes ownership of the file_chunk buffer, so it may either reuse the buffer for the memory chunk or allocate a new buffer and free file_chunk. Also returns the size of the returned buffer in mem_chunk_size. If disable_filters is set then filters are not applied to file_chunk.

6.2.5 mem_to_file

```
typedef herr_t(*H5D_chunk_format_mem_to_file_t)(void *chunk_format_info, void
*mem_chunk, void **file_chunk, hsize_t *file_chunk_size, hbool_t disable_filters,
hbool_t free_mem, hbool_t *free_file);
```

This function translates a memory format chunk into a file format chunk. If free_mem is set, frees the memory chunk afterwards. It should set free_file to FALSE if the file chunk should not be freed afterwards (i.e. if the buffer was reused and free_mem was not set), or TRUE otherwise. file_chunk may initially contain a pointer to a buffer of size file_chunk_size which may be used or freed by this function. This function should return the file chunk in file_chunk and the file chunk size in file_chunk_size.

6.2.6 mem_free

```
typedef herr_t (*H5D_chunk_format_mem_free_t)(void *chunk_format_info, void
*mem_chunk);
```

This function frees a memory format chunk.

6.2.7 file_free

```
typedef herr_t (*H5D_chunk_format_file_free_t)(void *chunk_format_info, void
*file_chunk);
```

This function frees a file format chunk.

6.2.8 mem_read

```
typedef herr_t (*H5D_chunk_format_mem_read_t)(void *chunk_format_info, const void
*mem_chunk, void *buf, const H5S_t *chunk_space, const H5S_t *buf_space);
```

This function reads the selected data from a memory format chunk into the selected region of a memory buffer.

6.2.9 mem_write

```
typedef herr_t (*H5D_chunk_format_mem_write_t)(void *chunk_format_info, void
*mem_chunk, const void *buf, const H5S_t *chunk_space, const H5S_t *buf_space);
```

This function writes the selected region of a memory buffer to the selected region of a memory format chunk.

6.2.10 file_read

```
typedef herr_t (*H5D_chunk_format_file_read_t)(void *chunk_format_info, haddr_t
chunk_offset, hsize_t chunk_size, void *buf, const H5S_t *chunk_space, const H5S_t
*buf_space);
```

This function reads the selected data from a chunk on disk directly to the selected region of a memory buffer. This function is optional and will not be supported by sparse datasets. For filtered datasets, this function will assume the chunk is unfiltered. In this case the upper level chunk code will only call this function if the chunk is unfiltered due to the edge chunk case.

6.2.11 file_write

```
typedef herr_t (*H5D_chunk_format_file_write_t)(void *chunk_format_info, haddr_t
chunk_offset, hsize_t chunk_size, const void *buf, const H5S_t *chunk_space, const
H5S_t *buf_space);
```

This function writes the selected data from a memory buffer directly to the selected region of a chunk on disk. This function is optional and will not be supported by sparse datasets. For filtered datasets, this function will assume the chunk is unfiltered. In this case the upper level chunk code will only call this function if the chunk is unfiltered due to the edge chunk case.

6.2.12 defined

```
typedef H5S_t *(*H5D_chunk_format_defined_t)(void *chunk_format_info, const void
*mem_chunk, const H5S_t *chunk_space);
```

This function returns a selection containing the defined elements in the supplied selection in a memory chunk. This function is optional and will only be supported by sparse chunks. If this callback is NULL the upper level will simply return the selection supplied by the app, since all elements are defined. If this callback is not NULL the upper level will assume unallocated chunks are completely undefined.

6.2.13 erase

```
typedef herr_t (*H5D_chunk_format_erase_t)(void *chunk_format_info, void
*mem_chunk, const H5S_t *chunk_space);
```

This function causes the selected elements to be undefined in the supplied memory chunk. This function is optional and will only be supported by sparse chunks. If this callback is NULL then attempting to use this functionality results in an error. If the chunk is not allocated and the callback is not NULL the upper level will simply skip this chunk.

6.2.14 fill

```
typedef void *(*H5D_chunk_format_fill_t)(void *chunk_format_info);
```

This function should create a memory chunk completely filled with the dataset's fill value. This function is optional and will not be defined for sparse chunks, since early allocation defeats the purpose of sparse datasets.

6.2.15 copy

```
typedef herr_t(*H5D_chunk_format_copy_t)(void *chunk_format_info, H5F_t *src_file,
void *mem_chunk);
```

This function copies the supplied memory chunk in place, which originates from the given source file (if src_file is NULL, it is from the same file). This should do things like copy and retarget vlens, and fix references, and should not change the size of the memory chunk. This function is used when allocating chunks with fill values, and when copying a dataset with H5Ocopy.

6.2.16 space_type

```
typedef herr_t(*H5D_chunk_format_space_type_t)(void *chunk_format_info, const H5S_t
*space, MPI_Datatype *new_type, int *count, hbool_t *is_derived_type, hbool_t
do_permute, hsize_t **permute_map, hbool_t *is_permuted);
```

This function creates an MPI datatype of the selected data in memory for file I/O. Will only be defined for formats without variable size, others will perform parallel I/O only on full chunks. The chunk format must also be able to subset within the chunk without being able to inspect it.

Implementation of this function could be skipped for the first pass, and instead hard code the parallel code to continue using H5S_mpio_space_type as it does currently. The presence of this function will improve code compartmentalization.

6.2.17 file_buf_free

```
typedef herr_t (*H5D_chunk_format_file_free_t)(void *chunk_format_info, void
*file_buf);
```

This function frees a file buffer allocated by space_type.

6.3 Code Flow

Much of the existing chunk code can be carried over with little to no modification. In particular the code in the io_init functions that creates selections for each chunk in memory and in the file can be carried over. The chunk cache could be reused (with some modification) or could be replaced with a better implementation. The way some chunk functionality uses the chunk format calls is described below.

Most of the cases described below take place for a single chunk. For these, io_init, or a similar routine, will have been run beforehand to generate a list of chunks involved in the operation and generate selections specific to each chunk. The chunk code will then iterate over these chunks and make the calls described for each one.

6.3.1 Read (Skip Cache)

For a read that bypasses the cache, the upper level chunk code first checks for the presence of the file_read call in the chunk format struct. If it is present, and either the dataset is not filtered or the

chunk is a partial edge chunk, it calls file_read, which completes the I/O operation. Otherwise, the chunk code finds the address and size of the chunk using the index, then calls file_alloc to allocate a buffer for the memory image of the chunk in the file. Next, it reads the chunk into the buffer, the calls file_to_mem, which converts the file image to a memory chunk. Finally it calls mem_read to read the selected data from the memory chunk to the application buffer, then calls mem_free to release the memory chunk.

6.3.2 Write (Skip Cache)

For a write that bypasses the cache, the upper level chunk code first checks for the presence of the file_write call in the chunk format struct. If it is present, and either the dataset is not filtered or the chunk is a partial edge chunk, it calls file_rw, which completes the I/O operation. Otherwise, the chunk code checks the index for the presence of the chunk on disk and checks for a full overwrite.

If the chunk is found on disk and it is not a full overwrite, the chunk code finds the address and size of the chunk using the index, then calls file_alloc to allocate a buffer for the memory image of the chunk in the file. Next, it reads the chunk into the buffer, then calls file_to_mem, which converts the file image to a memory chunk. Then it calls mem_write to write the selected data from the application buffer to the memory chunk. It then calls mem_to_file with no file_chunk and with free_mem set to TRUE. Finally it uses existing functionality to reallocate the chunk, write it, and update the index, then calls file_free if mem_to_file returned free_file as TRUE.

If the chunk is not found on disk or it is a full overwrite, the chunk code first calls fill if it is not a full overwrite or calls mem_alloc if it is. Next, it calls mem_write to fill the memory chunk with the data to write. Then it calls mem_to_file with no file_chunk and with free_mem set to TRUE. Finally it uses existing functionality to insert the chunk into the index, allocate it, and write it, then calls file_free if mem_to_file returned free_file as TRUE.

6.3.3 Load Into Cache

To load a chunk into cache, the chunk code first uses the index to find the address and size of the chunk, then calls file_alloc to allocate a buffer for the memory image of the chunk in the file. Next, it reads the chunk into the buffer, then calls file_to_mem, which converts the file image to a memory chunk. The chunk cache may now save mem_chunk in the cache, and keep track of the space used in the cache using the returned value mem_chunk_size.

6.3.4 Flush From Cache (No Evict)

To flush a chunk from the cache without evicting it, the chunk code first calls mem_to_file with no file_chunk and with free_mem set to FALSE. Then it uses existing functionality to insert the chunk into the index, allocate it, and write it, then calls file_free if mem_to_file returned free_file as TRUE. If variable_size is FALSE and the chunk was already present in the file, there is no need to update the index or allocate space for the chunk.

6.3.5 Flush From Cache (With Evict)

To flush a chunk from the cache and evict it, the chunk code first calls mem_to_file with no file_chunk and with free_mem set to TRUE. Then it uses existing functionality to insert the chunk into the index,

allocate it, and write it, then calls file_free if mem_to_file returned free_file as TRUE. Finally it updates its internal structures to reflect the fact that the chunk is no longer present in the cache. If variable_size is FALSE and the chunk was already present in the file, there is no need to update the index or allocate space for the chunk.

6.3.6 Parallel I/O

To perform parallel I/O, the chunk code first determines, using existing functionality, whether collective I/O is possible. If using independent I/O, the code flow is similar to the methods described in the serial case above. For chunk formats with fixed size chunks, the library will use existing functionality except calls to H5S_mpio_space_type will be replaced with calls to space_type.

For chunk formats with variable size chunks the library will use the existing functionality created for handling collective filtered I/O, except code that currently calls H5Z_pipeline will be changed to instead call file_to_mem or mem_to_file as appropriate. In addition, code that uses H5D_scatter_mem and H5D_gather_mem to manipulate data in the memory buffer will be replaced with calls to mem_read and mem_write as appropriate.

Note that independent I/O will be disallowed for chunk formats with variable size chunks. In addtition, the chunk cache will be disabled when the file is open for write access,

6.3.7 Set Extent

To be written

6.3.8 Early Allocation

To be written

6.3.9 Object Copy

To be written

6.4 Possible Additions

6.4.1 Unfiltered Edge Chunks

The current chunk implementation allows filtered datasets with filters disabled on partial edge chunks. It is thought that this is not useful for sparse datasets since it does not allow fast appends without read/modify/write on edge chunks, as it does for dense datasets. However, it would not be difficult to implement if a use case appears in the future for this. In this case, we would probably need to add a field “variable_size_unfiltered” which would be TRUE for sparse datasets and FALSE otherwise. The chunk code would then need to take this into account when allocating or modifying partial edge chunks.

6.4.2 Scratch Pad Buffer

In order to reduce the overhead associated with memory allocation, we may wish to add the ability to use a scratch pad buffer that persists between calls to the chunk format layer. This would be especially useful in cases where the buffer doesn’t fit neatly into a free list object. This could be implemented in a few different ways. It could be handled by the upper layer, with the buffer and buffer size passed to the

format layer through function parameters, or it could be handled by the format layer, which would declare the scratch pad size so it could be counted by the chunk cache, and free the scratch pad when the chunk cache wants to make room.

6.4.3 Parallel Collective Late Allocation

Full implementation of the chunk format layer should allow fairly easy implementation of late allocation in collective parallel I/O. This is necessary for sparse datasets, and that implementation could be extended to other cases. This would impact performance in the dense unfiltered case, since all ranks would need to coordinate allocation, but the tradeoff may be worth it in some cases.

6.5 Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract Number DE-AC02-05CH11231.