RFC: Programming Model to Support Sparse Data in HDF5H5 (Draft)

John Mainzer (john.mainzer@lifeboat.llc**)**
Elena Pourmal (elena.pourmal@lifeboat.llc**)**[1]
Vailin Choi (vchoi@hdfgroup.org **)**

We propose to extend HDF5 File Format, C library and command-line tools to support sparse arrays in HDF5. The new storage is based on the concept of the "Structured Chunk" that allows us to keep element locations and their values together. The concept can be applied to store other types of variable-size data, for example, variable-length strings and non-homogeneous arrays. File Format extensions for sparse data are discussed in [4].

This document focuses on programming model, new and existing APIs to manage sparse data in HDF5, and changes to the command-line tools.

The structure of the document as follows. Section 1 provides background information on sparse data. Section 2 discusses the programming model and new APIs to support sparse arrays in HDF5. *The summary listing of the new proposed functions can be found in Appendix to this document.* Section 3 has a brief discussion of the HDF5 sparse storage vs. existing memory optimized formats for representing sparse matrices.

The purpose of the document is to kick-off discussion of the new sparse feature. The document will be updated as feedback is received from the HDF5 users' community and as new use cases are discovered.

The extensions to the HDF5 File Format [4] and the new APIs proposed in this document to support sparse storage will be contributed to the open source HDF5 software maintained by The HDF Group.

Doxygen documentation for proposed functions was created and available for review from https://gamma.hdfgroup.org/ftp/pub/outgoing/vchoi/SPARSE/hdf5lib_docs/html/_r_m.html. Current implementation of the proposed public functions (version 19 of this RFC) is available in the feature branch

https://github.com/HDFGroup/hdf5/tree/feature/sparse_data.

***Update for version 20-21:** This version was modified to capture proposals and comments from the HDF5 community members. Programming model and public APIs will be reconsidered after the first prototype is finished and released. Section 2.4 and H5Pset(get)_density and H5Dnative* functions were contributed by Quincey Koziol (koziol@nvidia.com). Summary of the proposed changes that sparse prototype*

[1] Contact person

implementation will follow can be found in "RFC: Finalizing Sparse Data Programming Model and APIs".

This document is work in progress. **Programming model and Public APIs are subject to change.**

1 Introduction

HDF5 was designed for "dense" array storage, where each element of a data array is mapped to a location in HDF5 file. Sometime storing every element of the array is disadvantageous. For example, in the case of experimental and observational data, elements with *useful or "defined"* data are rare[2] and only they have to be stored. Another example is a data array with a repetitive element value. Figure 1 shows a matrix with more than half of its elements being "0".

We call data in the provided examples "sparse" data. Support for efficient and portable storage of sparse data in HDF5 is a long-standing request from the HDF5 user community.

Figure 1: Example of a sparse matrix

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	66	69	72	75	78	81	0	0
0	0	96	99	102	105	108	111	0	0
0	0	126	129	132	135	138	141	0	0
0	0	0	0	0	0	0	0	0	2
100	0	-100	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	3	0

When "dense" storage is used to store the matrix in an HDF5 dataset, there is no distinction between "defined" and "non-defined" elements (represented by 0 values in the example above). While storage savings can be achieved by using compression and writing only selections that contain non-zero elements, locations of defined elements in the matrix are not saved. If there is a desire to have a quick access to defined elements one has to save coordinates of the defined elements along with the matrix data.

While there are many ways of representing sparse data in HDF5 (see Section 3 for further discussion), currently, there is no a standard way for storing sparse arrays in HDF5. Current common practice is to store two one-dimensional arrays – one containing defined elements and another one containing

[2] In the use cases we cited in the provided references only 0.1% to 10% of gathered data is of interest, but it may contain a bigger percentage. We leave this quantification to the HDF5 user. When the data is stored in HDF5 "non-useful" data is usually represented by a fill value (default is 0). Please notice the difference between "non-useful" data and missing data, i.e., the data that is not in the array though one expects them to be present (e.g., earth surface temperature was not detected because of cloud covering). Currently, HDF5 doesn't have built-in capabilities to support "useful", "non-useful" and missing data. Implementation of this capability is left to a user. Sparse storage targets storage of "useful" data and elimination of "non-useful" data.

indices of these elements. In general, there are many variations on sparse data organization in HDF5 depending on application's needs.

Our proposed implementation offers sparse array storage that is independent from in-memory representation of the sparse data thus offering sparse data portability between applications. It also requires minimal changes to applications' codes as we show in the next sections.

For initial implementation ideas for sparse storage and examples of sparse data in HDF5 we refer the reader to the RFC document [1][3] and the "Sparse Data Management in HDF5" paper [2]. The documents provide motivation for adding support for sparse data to HDF5, discuss use cases and design options for storing sparse data in HDF5 in detail. In this RFC we focus on programming model and discuss APIs needed to manage sparse data.

Based on the ideas described in the original RFC, we introduce new type of storage called *structured*[4] *chunk storage* that allows storage of multiple, frequently variable-length sections of data in a chunk. See [4] for the structured chunk layout and other file format extensions in support for structured chunk data. Here we give a brief introduction to the structured chunk concept using sparse data example.

As in the existing chunked storage, an array is divided into equally-sized logical chunks as shown on

Figure 2, but chunk elements are stored according to the type of data (e.g., sparse). In the case of sparse data, locations of the "defined" elements are stored as an encoded selection along with the elements' values as shown in Table 1.

[3] We will refer to this document as "original RFC".

[4] It was proposed to use "sectioned" instead of "structured" adjective for new storage or a chunk with multiple sections. In this version of RFC, we didn't use the new term (the latest File Format Spec https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/HDF5%20File%20Format%20Specification%20Version%203.1%202025-01-13-RFCv22.html does). We will update API names when we finalize the programming model and APIs. APIs functionality is reflected in this version and is not expected to go through substantial change.

Figure 2: An array is divided into 4 by 5 chunks with the last two chunks having "ghost zones" (as for regular chunked storage). For each chunk only non-zero values and their locations are stored in the structured chunk.

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	66	69	72	75	78	81	0	0
0	0	96	99	102	105	108	111	0	0
0	0	126	129	132	135	138	141	0	0
0	0	0	0	0	0	0	0	0	2
100	0	-100	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	3	0

Table 1: Conceptual layout of structured chunk for storing sparse data of fixed-size datatype.

Section 0: Encoded Selection
Section 1: Data

For each logical chunk we will store an Encoded Selection (encoded matrix coordinates) and values of defined elements (Data); for example, for the upper-left chunk encoded coordinates and the values 66, 69, 72, 96, 99 and 102 (shaded on

Figure 2) are stored (see Table 2); for the low-right chunk it would be encoded coordinates of value 3 and the value 3 itself.

Table 2: Conceptual view of upper-left sparse chunk

Encoded Selection for [2,2]-[3,4] hyperslab
Data 66 69 72 96 99 and 102

Please note that in our example data has fixed-size datatype and as a result, a structured chunk will have only two sections. If data has variable-length datatype (e.g., strings) we will need an additional section to store lengths and references to the elements stored in a data section (see

Table 3).

Table 3: Conceptual layout of structured chunk for storing sparse data of variable-length datatype.

Section 0: Encoded Selection
Section 1: Fixed-size data (exact format TBD; e.g., lengths and references to data in Section 2)
Section 2: Variable-length Data

In the future we will be able to use structured chunk storage to re-implement variable-length raw data in HDF5. Such chunk will not have Encoded Selection section but will have two other sections as shown in

Table 3.

The concept of a structured chunk allows us to implement more complex objects, for example, non-homogeneous arrays and superpose sparse storage with non-homogeneous arrays.

We should also note here that the design enables filtering of different sections of a structured chunk using different filter pipelines. In case of the sparse structured chunk for a dataset with a fixed-size datatype, one could apply different filter pipelines to the Encoded Selection Section and to the Data

Elements section. In case of the sparse structured chunk for a dataset with a variable-length datatype, one could apply different filter pipelines to each of three sections (see

Table 3)

As a reader will see from the discussion in the following sections, data organization in the structured chunk is hidden from the user unless direct chunk I/O is used (see 2.2).

2 HDF5 Programming Model for Sparse Data

This section discusses programming model for sparse data.

Usage of structured chunk storage will follow the standard HDF5 programming model with minimal changes to the code.

Here we outline the steps required to create, write, open and read sparse datasets and flag the steps that will be different from the current work flow when writing and reading HDF5 "dense"[5] datasets.

On create and write the following steps below are performed. Please notice that these are the regular steps to create and write an HDF5 dataset.

1. Define dataset creation property to use structured chunk storage for sparse data using the `H5Pset_struct_chunk` or `H5Pset_sparse_chunk` function[6].
2. Define other applicable creation properties, for example, compression using the **new function** `H5Pset_filter2` or one of the predefined compression functions (e.g., `H5Pset_deflate`).
3. Create a dataset using the defined properties.
4. Define a selection with the data values in the memory space.
5. Define a selection in the file indicating where to write the "*defined*" data.
   Note: This step is the same as writing a selection of a dense array, but *underlying storage is different.*
6. Write data buffer.
7. Close the dataset.
8. Close property lists.

On the read the following steps are performed.

1. Open a sparse dataset.
2. Optional:
   a. Get chunk size and other creation properties including filtering. See 2.3 for the new filtered structured chunks functions.
3. Select elements in the dataset in the file by using
   a. Current hyperslab selection API (i.e., no changes for read are required)
      or
   b. Get selection of "*defined*" elements in the provided bounding box using the **new function** `H5Dget_defined`.
4. Use the function `H5Sget_select_npoints` to find the number of the elements in the current selection and to allocate a buffer of appropriate size to read the selected data.
5. Read selected data.

[5] The HDF5 library requires that *all* elements of the dataset are defined with user-supplied values or fill-values, and it treats data as "dense", mapping each data element to storage during I/O operations.

[6] We should also decide on what will be the behavior of the calls to the existing APIs `H5Pset_layout` with **new storage layout** H5D_STRUCT_CHUNK, H5D_SPARSE, and `H5Pset_chunk` (initial proposal to avoid new API; for now, we decided to use a new API). Please also see a note in section 2.1.3.

6. Closed the dataset

As one can see there are no differences how HDF5 handles data stored using structured chunk storage vs. any other types of storage. We think it would be useful to have an "erase" API to "undefine" elements in a sparse array. Please notice that new elements can be always added and current elements can be modified using the current APIs by explicitly re-writing data.

2.1 APIs to Handle Sparse Data[7]

In this section we provide descriptions of new APIs to handle sparse data.

2.1.1 H5Pset_struct_chunk

Sets structured chunked storage. The storage is used for sparse data of any datatype, and for dense data with variable-length datatype.

Signature

herr_t H5Pset_struct_chunk (*hid_t* plist_id, *int* ndims, *const hsize* dim[], *unsigned* flag)

Parameters

plist_id	IN: Dataset creation property identifier
ndims	IN: The number of chunk dimensions
dim	IN: An array defining the size, in dataset elements, of each chunk
flag	IN: Flag that indicates structured chunk storage type. Current types include sparse chunk storage type and variable-length chunk storage type.

Description

H5Pset_struct_chunk() sets structured chunk storage layout, chunk sizes and a type of structured chunk storage for a dataset. This function is only valid for dataset creation property lists.

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

The values of the dim array define the size of the chunks. The unit of measure for dim values is in dataset elements.

As a side-effect of this function, the creation property is modified to H5D_STRUCT_CHUNK storage layout, if it was previously set using H5Pset_layout[8] function with any other storage layout type.

[7] API names and signatures are subject to change.

[8] We should add new layout to the H5Pset/get_layout function descriptions

The value of the `flag` parameter can be H5D_SPARSE_CHUNK to store sparse data of any datatype or H5D_VL_CHUNK[9] to store dense data of variable-length datatype.

Returns

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

2.1.2 H5Pget_struct_chunk

Retrieves the size of chunks and the structured chunk storage type for the raw data of a dataset with structured chunk layout.

Signature

```
int H5Pget_struct_chunk (hid_t plist_id, int max_ndims, const hsize dim[],
unsigned *flag)
```

Parameters

`plist_id`	IN: Dataset creation property identifier
`max_ndims`	OUT: The number of chunk dimensions
`dim`	OUT: An array defining the size, in dataset elements, of each chunk
`flag`	OUT: Flag that indicates structured chunk storage type. Current types include sparse chunk storage type and variable-length chunk storage type.

Description

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

`H5Pget_struct_chunk()` retrieves the size of chunks and the structured chunk storage type for the raw data of a structured chunk layout dataset. This function is only valid for dataset creation property lists. At most, `max_ndims` elements of dim will be initialized.

The type of structured chunk storage used will be retrieved in flag. The value can be `H5D_SPARSE_CHUNK` for storing sparse data of any datatype or `H5D_VL_CHUNK` for storing dense data of variable-length datatype.

2.1.3 H5Pset_sparse_chunk

Sets chunked storage for sparse data.

Signature

```
herr_t H5Pset_sparse_chunk (hid_t plist_id, int ndims, const hsize dim)
```

[9] The names of the flags are subject to change.

Parameters

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

Description

`H5Pset_sparse_chunk()` sets the size of the chunks used to store a sparse dataset. This function is only valid for dataset creation property lists.

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

The values of the `dim` array define the size of the chunks. The unit of measure for `dim` values is in dataset elements.

As a side-effect of this function, the creation property is modified to H5D_STRUCT_CHUNK storage layout, if it was previously set using H5Pset_layout function with any other storage layout type.

Returns

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

Note: We don't recommend to implement this function and always use H5Pset_struct_chunk instead. If we add a specific set/get functions for sparse storage type, we will need to introduce set/get functions for variable-length storage type, and for any other storge type that will be using structure chunks. Therefore, we do not recommend this function in order to keep the number of HDF5 APIs under control.

We should also note here that the programming model can be implemented by using existing APIs `H5Pset_layout` with `H5D_SPARSE_CHUNK` or `H5D_VL_CHUNK` storage type that would assume structured chunk layout, and the `H5Pset_chunk` function to set chunk sizes.

> **Comment:** *I strongly advocate for this as the solution for creating sparse, VL, etc. sub-types of chunked storage. Users should _not_ be required to understand the implementation of how storage is optimized for their particular storage pattern.*
>
> *This is particularly true in the current timeframe where VOL connectors can choose to implement sparse storage in some other way than structured chunks.*
>
> *We *definitely* should not include the current implementation of optimizations for storing sparse in the API, which has to last a very long time.*
>
> **Comment:** `H5D_VL_CHUNK` *flag was introduced to allow users to have both ways of storing VL data. If they specify the flag, then new storage format is used. Please notice that if sparse data*

is VL, one just need to use H5D_SPARSE_CHUNK since the library will know about VL type anyway.

2.1.4 H5Pset_density

Sets expected density of a dataset.

Signature

`herr_t H5Pset_density (hid_t plist_id, unsigned type)`

Parameters

`plist_id`	IN: Dataset creation property identifier
`type`	IN: Enumerated value that indicates chunk density type[10]. Current types include dense and sparse.

Description

H5Pset_density sets the expected storage density for a dataset. This function is only valid for dataset creation property lists.

The currently defined density types are: DENSE (existing behavior) and SPARSE for datasets that are known in advance to contain sparsely written dataset elements.

Advice to implementors

How to optimize sparse vs. dense storage is up to the implementation. From a user's perspective, they shouldn't need to think about the optimization technique, just their view of the world. So, if they say "sparse", structured chunks or some other mechanism can be used to get them the result they desire. Variable-length structured chunks are also an optimization that the library can choose at creation time, based on the dataset's datatype. Dense vs sparse can't be inferred from other parameters and the user must hint them directly. All the other information is already available in existing API routines.

Returns

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

It is assumed that there will be `H5Pget_density` that retrieves the density of a dataset.

[10] A more flexible pair of parameters might be ('x' and 'y'), where the application is indicating that 'x' elements out of 'y' total elements are estimated to be written (i.e. a percentage, expressed as a fractional density). That would allow for a spectrum from dense to sparse and give more room for implementors to optimize for a variety of densities.

H5Dget_defined

Retrieves a dataspace selection containing defined elements.

Signature

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

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

Description

`H5Dget_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_CHUNK`. For other layouts this function will simply return a copy of `file_space_id`, as all elements are defined for non-sparse datasets.

Returns

Returns a dataspace with a selection containing all defined elements that are also selected in `file_space_id` if successful; otherwise returns H5I_INVALID_HID.

2.1.5 H5Derase

Deletes elements from a dataset.

Signature

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

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

Description

`H5Derase` deletes 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 `H5Dget_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 structured chunk layout of the `H5D_SPARSE_CHUNK type`. For other layouts this function will return an error.[11]

Returns

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

2.2 APIs to Support Direct Chunk I/O

In this section we outline the new APIs to allow direct chunk I/O on non-filtered and filtered structured chunks. They are similar to the corresponding chunk functions for dense storage (see the existing H5D*chunk* functions). As in the case of the dense storage, it is user's responsibility to provide correct information about the structured chunk on write and use information returned by the library to access the data stored in the structured chunk.

We propose new functions `H5Dwrite_struct_chunk` and `H5Dread_struct_chunk` for direct chunk I/O, `H5Dget_struct_chunk_info` and `H5Dget_struct_chunk_info_by_coord` to get information about the structured chunk, and `H5Dstruct_chunk_iter` to iterate over the structured chunks. For reader's convenience the Table below lists existing functions for dense chunks and proposed new counterparts for the structured chunks.

Dense chunk functions	Structured chunk functions[12]	Comment
H5Dwrite_chunk	H5Dwrite_struct_chunk	See section 2.2.1
H5Dread_chunk	H5Dread_struct_chunk	See section 2.2.2
H5Dget_chunk_info	H5Dget_struct_chunk_info	See section 2.2.3
H5Dget_chunk_info_by_coord	H5Dget_struct_chunk_info_by_coord	See section 2.2.4
H5Dchunk_iter	H5Dstruct_chunk_iter	See sections 2.2.5 and 2.2.6
H5Dget_chunk_storage_size H5Dget_num_chunks	-	No special function provided for structured chunk. See discussion in Section 2.2.7

Structured chunk info is represented by the following data structure that all the functions above will take or return as a parameter:

```
typedef struct H5D_struct_chunk_info_t {
    enum [13]        type;[14]              /* Type of the structured chunk;        */
```

[11] One can argue that this function should be extended to work with other types of storage, writing fill values to the selected elements. It is a convenience function.

[12] I advocate for versioning existing functions; it will be one set of functions to handle both dense and sparse chunks.

[13] We can define enum type to list mnemonics for different sections typedef enum H5_section_type { H5_SECTION_UNKNOWN = -1, H5_SECTION_SELECTION, H5_SECTION_FIXED, H5_SECTION_VL}

[14] To make output more descriptive, we can return array that contains enums for each section, i.e., H5_SECTION_SELECTION, H5_SECTION_FIXED, but we have a problem here…since we also need a knowledge of datatype

```
                                    /* currently H5D_SPARSE_CHUNK             */
    uint8_t      num_sections;      /* Number of sections in structured chunk */
    uint16_t     filter_mask[];     /* Array of num_sections size             */
                                    /* Contains filter mask for each section. */
                                    /* It is 0 when no filters are applied.   */
    size_t       section_size[];    /* Array of num_sections size             */
                                    /* Contains the size of each section      */
    size_t       section_orig_size[]; /* Array of num_sections size           */
                                    /* Contains original size of each section */
} H5D_struct_chunk_info_t
```

Please notice that in unfiltered case the values of the sections' sizes will be the same as the corresponding original sizes. The values are 0 if the section is empty. See File Format Specification, section IX, Appendix E: Layout of Structured Chunk.

2.2.1 H5Dwrite_struct_chunk

Writes structured chunk.

Signature

```
herr_t H5Dwrite_struct_chunk (hid_t dset_id,
                              hid_t dxpl_id,
                              H5D_struct_chunk_info_t *chunk_info,
                              const hsize_t *offset
                              void *buf[])
```

Parameters

dset_id	IN: Dataset identifier
dxpl_id	IN: Data transfer property list identifier
chunk_info	IN: Information about the structured chunk
offset	IN: Logical position of the chunk's first element in the array
buf	IN: Array of pointers to the sections of the structured chunk. The size of the array is equal to the number of sections in the structured chunk.

Description

H5Dwrite_struct_chunk writes a structured chunk specified by its logical offset offset to dataset dset_id. The HDF5 library assembles the structured chunk according to the information provided in the chunk_info parameter and using data pointed by buf. buf is an array of pointers to the buffers containing data for each section of the structured chunk. Initially, the function will support only sparse chunks of the fixed-size data. Such chunks have only two sections: the first one contains encoded selection[15] for defined elements and the second one contains the values of defined elements.

[15] Encoded selection is a binary blob returned in the buf parameter of the H5Sencode2 function

Returns

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

2.2.2 H5Dread_struct_chunk

Reads structured chunk.

Signature

```
herr_t H5Dread_struct_chunk (hid_t dset_id,
                             hid_t dxpl_id,
                             const hsize_t *offset,
                             H5D_struct_chunk_info_t *chunk_info,
                             void *buf[])
```

Parameters

dset_id	IN: Dataset identifier
dxpl_id	IN: Data transfer property list identifier
offset	IN: Logical position of the chunk's first element in the array
chunk_info	IN/OUT: Information about the structured chunk
buf	IN/OUT: Array of pointers to the sections of structured chunk. The size of the array is equal to the number of sections in the structured chunk. NULL can be passed to return chunk information, specifically, the number of sections and the sizes of each section.

Description

H5Dread_struct_chunk reads a structured chunk as specified by its logical offset offset in a chunked dataset dset_id and places data into the provided buffers pointed by buf. Information about the structured chunk is returned via the chunk_info parameter. buf is an array of pointers to the buffers into which data for each section of the structured chunk will be read into. It is application's responsibility to allocate buffers of the appropriate size. Initially, the function will support only sparse chunks of the fixed-size data. Such chunks have only two sections: the first one contains encoded selection for defined elements and the second one contains the values of defined elements.

Returns

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

2.2.3 H5Dget_struct_chunk_info

Gets structured chunk info using chunk index.

Signature

```
herr_t H5Dget_struct_chunk_info (hid_t dset_id,
```

```
                                    hid_t fspace_id,
                                    hsize_t chunk_idx,
                                    const hsize_t *offset,
                                    H5D_struct_chunk_info_t *chunk_info,
                                    haddr_t *addr,
                                    hsize_t *chunk_size)
```

Parameters

dset_id	IN: Dataset identifier
fspace_id	IN: File dataspace selection identifier
chunk_idx	IN: Chunk index
offset	OUT: Logical position of the chunk's first element in the array
chunk_info	OUT: Information about the structured chunk
addr	OUT: Chunk address in the file
chunk_size	OUT: Chunk size in bytes; 0 if the chunk does not exist

Description

H5Dget_struct_chunk_info retrieves the offset coordinates, offset, structured chunk information chunk_info, chunk's address, addr, and the size, chunk_size, for the dataset specified by the identifier dset_id and the chunk specified by the index, chunk_idx. The chunk belongs to a set of chunks in the selection specified by fspace_id. If the queried chunk does not exist in the file, the size will be set to 0 and address to HADDR_UNDEF. NULL can be passed in for any OUT parameters.

chunk_idx is the chunk index in the selection. The index value may have a value of 0 up to the number of chunks stored in the file that have a nonempty intersection with the file dataspace selection.

Returns

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

2.2.4 H5Dget_struct_chunk_info_by_coord

Gets structured chunk info using chunk coordinates.

Signature

```
herr_t H5Dget_struct_chunk_info_by_coord (hid_t dset_id,
                                          const hsize_t *offset,
                                          H5D_struct_chunk_info_t *chunk_info,
                                          haddr_t *addr,
                                          hsize_t *chunk_size)
```

Parameters

dset_id	IN: Dataset identifier
offset	IN: Logical position of the chunk's first element in the array
chunk_info	OUT: Information about the structured chunk
addr	OUT: Chunk address in the file
chunk_size	OUT: Chunk size in bytes; 0 if the chunk does not exist

Description

H5Dget_struct_chunk_info_by_coord retrieves the structured chunk information chunk_info, chunk's address, addr, and the size, chunk_size, for the dataset specified by the identifier dset_id and the chunk specified by the its coordinates, offset. If the queried chunk does not exist in the file, the size will be set to 0 and address to HADDR_UNDEF. The value pointed to by chunk_info will not be modified. NULL can be passed in for any OUT parameters.

offset is a pointer to one-dimensional array with a size equal to the dataset's rank. Each element is the logical position of the chunk's first element in a dimension.

Returns

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

2.2.5 H5Dstruct_chunk_iter

Iterates over all structured chunks.

Signature

```
herr_t H5Dstruct_chunk_iter (hid_t dset_id,
                             hid_t dxpl_id,
                             H5D_struct_chunk_iter_op_t cb,
                             void *op_data)
```

Parameters

dset_id	IN: Dataset identifier
dxpl_id	IN: Property list identifier
cb	IN: Call back function provided by user; called for every chunk
op_data	IN: User-defined pointer to data required by callback function

Description

H5Dstruct_chunk_iter iterates over all structured chunks in the dataset, calling the user specified callback function, cb, and callback's required data, op_data.

Returns

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

2.2.6 Callback for H5Dstruct_chunk_iter

```
typedef int(*H5D_struct_chunk_iter_op_t) (const hsize_t *offset,
                                          H5D_struct_chunk_info_t *chunk_info,
                                          haddr_t *addr,
                                          hsize_t *chunk_size,
                                          void *op_data)
```

Parameters

offset	IN: Logical position of the chunk's first element in the array
chunk_info	IN: Information about the structured chunk
addr	IN: Chunk address in the file
chunk_size	IN: Chunk size in bytes; 0 if the chunk does not exist
op_data	IN: User-defined pointer to data required by the callback function

Returns

- Zero (H5_ITER_CONT) causes the iterator to continue, returning zero when all elements have been processed.
- A positive value (H5_ITER_STOP) causes the iterator to immediately return that value, indicating short-circuit success.
- A negative (H5_ITER_ERROR) causes the iterator to immediately return that value, indicating failure.

2.2.7 Other considerations

The existing functions H5Dread_chunk, H5Dget_chunk_storage_size and H5Dget_num_chunks should work without changes on the structured chunks. It is application's responsibility to interpret data in the structured chunk when the H5Dread_chunk function is used. H5Dwrite_chunk cannot be used to write structured chunk because the function doesn't pass chunk's metadata that has to be stored in the chunk index. The same is true for H5Dget_chunk_info, H5Dget_chunk_info_by_coord, and H5Dchunk_iter since metadata information for structured chunk is more complex than can be passed to the current functions.

The behavior of the existing chunk functions when used with the structured chunk is summarized in the table below.

Dense chunk functions	Behavior on structured chunk
H5Dwrite_chunk	Fails
H5Dread_chunk	Fails

H5Dget_chunk_info	Fails
H5Dget_chunk_info_by_coord	Fails
H5Dchunk_iter	Fails
H5Dget_chunk_storage_size	Works as for dense chunk
H5Dget_num_chunks	Works as for dense chunk

2.3 Structured Chunk Filtering

Each section of the structured chunk contains data of a specific datatype. For example, for sparse dataset of floats, each structured chunk will have two sections: one for the encoded selection and another one that stores floating point data elements. Obviously, the same compression method may not be optimal on both sections, or may not be desired at all. The existing programming model allows specification of a filter pipeline for each individual section. Instead of adding new functions we propose to version the existing functions that manage HDF5 filters. See [6] for HDF5 API versioning approach. The new version 2 of the functions can be used with both current chunk storage (there is just one section) and structured chunk storage including sparse chunk. They also address deficiency of the current APIs for passing filter's data as the reader will see next.

2.3.1 H5Pset_filter2

The function adds a filter to the filter pipeline for a specified section of a sparse chunk. The function accepts new parameter `section_number` that specifies the section of the structured chunk to which the filter is applied. Please notices other differences with the existing `H5Pset_filter` function signature. The new signature addresses deficiencies of passing filter's data by using a `void` pointer to a buffer with an auxiliary data for the filter instead of the `unsigned int` data array. Data type for the `flags` parameter was changed to `uint64_t` to provide more flexibility to the VOL connectors that use the function. The new function can be used on both datasets and group creation property.

Signature

```
herr_t H5Pset_filter2 (hid_t plist_id, H5_section_type section_id,
                       H5Z_filter_t filter,
                       uint64_t flags,
                       size_t buf_size,
                       const void *buf)
```

Parameters

`plist_id`	IN: Object creation property list identifier
`section_number`	IN: An integer to specify section number. The value is 0 to 255 when native HDF5 file format is used. When applied to dense datasets, the flag is ignored. For the sparse chunk the convenience flag can be used to specify a section of the sparse chunk to be filtered as described below

	H5Z_FLAG_SPARSE_SELECTION: Adds the filter to the filter pipeline for the encoded selection section of the sparse chunk. It has the same effect as passing 0. The flag will be ignored if the structured chunk is not sparse. H5Z_FLAG_SPARSE_FIXED_DATA: Adds the filter to the filter pipeline for section 1 of the sparse chunk. It has the same effect as passing 1. H5Z_FLAG_SPARSE_VL_DATA: Adds the filter to the filter pipeline for section 2 of the sparse chunk if data has variable-length datatype. It has the same effect as passing 2.
filter	IN: Filter identifier for the filter to be added to the pipeline
flags	IN: Bit vector specifying certain general properties of the filter
buf_size	IN: Size in bytes of buf buffer
buf	IN: Buffer with an auxiliary data for the filter

Description

H5Pset_filter2 adds the specified filter identifier and corresponding properties to the end of an output filter pipeline for the section of the structured chunk specified by the section_number parameter. The parameter is an integer with the value 0 to 255 if native HDF5 file format is used.

plist_id is a dataset creation property identifier. The buffer buf of size buf_size contains auxiliary data for the filter. The values will be stored in the Structured Chunk Filter Pipeline message in the dataset object header as part of the filter information.

The flags argument is a bit vector with the fields specifying certain general properties of the filter as documented in the description of the current H5Pset_filter function.

Please note that H5Pset_edc_check function will be applicable to the structured chunk storage with enabled filtering but may not be available with the first release of the sparse feature.

2.3.2 Other extensions to manage structured chunk filters

We will need to provide the new versions of the existing functions to manage filter pipelines for structured chunk sections. The new versions mimic the signature of the existing functions with the changes similar to the changes done to the function H5Pset_filter2 signature. Below is the list of the proposed functions and their short descriptions. Parameters meaning stay the same as for the existing filter functions. Parameter section_name is a name of the chunk section provided by the library. We may need a function that returns names when structured chunk storage layout is set. But this can be done only when datatype is defined, i.e., instead of dcpl, we need to use dataset ID

```
H5Pget_struct_sections_info (dset_id, *number_of sections, *H5_section_type buf[]);
```

It can work with dense chunks too and VL data in old format (H5_SECTION_FIXED). The function may return H5_DENSE or special return value instead of error.

/* Returns the number of filters in the pipeline for a section of structured chunk */
```
herr_t H5Pget_nfilter2 (hid_t plist_id, ~~uint64_t~~ H5_section_type_t section_name,
int num_filters);
```

/* Returns information for a filter in the pipeline for a specified section */
```
H5Z_filter_t H5Pget_filter3 (hid_t dcpl, uint64_t section_name, unsigned idx,
uint64_t *flags, size_t *buf_size, void *buf, size_t namelen, char name[],
unsigned *filter_config);
```

/* Returns information for a filter specified by its identifier in the pipeline for a specified section of structured chunk */
```
H5Z_filter_t H5Pget_filter_by_id3 (hid_t dcpl, uint64_t section_name, H5Z_filter_t
filter, uint64_t *flags, size_t *buf_size, void *buf, size_t namelen, char name[],
unsigned *filter_config);
```

/* Removes a filter in the filter pipeline for a specified section */
```
herr_t H5Premove_filter2 (hid_t plist_id, uint64_t section_name, H5Z_filter_t
filter);
```

/* Modifies a filter in the filter pipeline for a specified section of structured chunk */
```
herr_t H5Pmodify_filter2 (hid_t plist_id, uint64_t section_name, H5Z_filter_t
filter, uint64_t flags, size_t buf_size, const void *buf);
```

2.3.3 Behavior of predefined filter functions

Predefined filter functions `H5Pset_deflate`, `H5Pset_scaleoffset`, `H5Pset_nbit`, `H5Pset_shuffle`, and `H5Pset_szip` can be used on the datasets with the structured chunk storage. When used, a filter will be added to the filter pipeline for each section of the structured chunk. For example, by calling `H5Pset_deflate (dcpl_t, 9)` for sparse dataset with fixed-size datatype the DEFLATE filter will be added to the filter pipeline for the encoded selection section (Section 0) and the fixed-size data section (Section 1). Using the function is also equivalent to calling `H5Pset_filter2` with the appropriate flags as shown in the code snippet below.

```
flags = H5Z_FLAG_OPTIONAL;
sec_num = H5Z_FLAG_SPARSE_SELECTION;[16]
status = H5Pset_filter2 (dcpl, sec_num, H5Z_FILTER_DEFLATE, flags, …);
sec_num = H5Z_FLAG_SPARSE_FIXED_DATA;
status = H5Pset_filter2 (dcpl, sec_num, H5Z_FILTER_DEFLATE, flags, …);
```

[16] These names should be section names (mnemonics) and are subject to change. The issue will be addressed in the next version of this RFC.

The H5Pset_deflate call for sparse dataset with variable-length datatype will add DEFALTE filter to the filter pipelines of each three sections of structured chunk. This is equivalent to calling H5Pset_filter2 with the appropriate flags as shown above and the third call shown in the code snippet below.

```
sec_num = H5Z_FLAG_SPARSE_VL_DATA;
status = H5Pset_filter2 (dcpl, sec_num, H5Z_FILTER_DEFLATE, flags, …);
```

One can remove a filter from the section's filter pipeline using H5Premove_filter2 function.

2.3.4 Note on H5Pset_fletcher32

Please notice that adding Fletcher32 to filter pipelines may not be practical for the structured chunk except when it is applied to the sections that contain raw data of the fixed-size datatype.

The HDF5 library automatically stores checksums for the sections of the structured chunk that contain data required to interpret the data stored in other sections. For example, for the sparse chunk the HDF5 library will always store the checksum for the encoded selection section (Section 0) since its corruption will cause loss of the defined elements' coordinates making impossible for the HDF5 library to read data elements into their locations in the sparse dataset. If sparse chunk contains variable-length data, then the HDF5 library will also store checksum for Section 1 containing fixed-size data required to find variable-length data in Section 2. If sparse chunk contains data of fixed-type datatype, then is will be stored in Section 1 of sparse chunk and Fletcher32 filter can be added to detect corruption of sparse dataset data elements.

2.4 Alternative APIs to handle direct chunk I/O

In this section we provide descriptions of new APIs to handle direct chunk I/O on all types of chunks. These routines are specific to the native VOL connector and are prefixed with 'native' accordingly.

The following typedefs are used for several routines:

```
typedef enum H5D_native_section_type_t {
      H5D_NATIVE_SECTION_SELECTION,        /* Selections, for sparse storage */
      H5D_NATIVE_SECTION_FIXED,            /* Fixed-size element data */
      H5D_NATIVE_SECTION_VLEN,             /* Variable-length element data */
} H5D_native_section_type_t;

typedef struct H5D_native_section_info_t {
      H5D_native_section_type_t  type;     /* Type of native section */
      uint16_t  filter_mask;               /* Filter mask for section */
      size_t   size;                       /* Size of section (in bytes) */
} H5D_native_section_info_t;
```

2.4.1 H5D~~native~~(chunk)_get_section_count

Queries the number of sections for a native dataset.

Signature

```
herr_t H5Dnative_get_section_count (hid_t dset_id,
                                    unsigned *num_sections)
```

Parameters

dset_id	IN: Dataset ID
num_sections	OUT: Number of sections for dataset

Description

H5Dnative_get_section_count() returns the number of sections used to store element values for a dataset with the native VOL connector. Legacy datasets have one section, but other optimizations are possible for sparse datasets and datasets with variable-length datatype storage.

Advice to users

This routine is used to determine the number of sections for calls to H5Dnative_get_section_types, H5Dnative_get_chunk_info, and H5Dnative_read_chunk.

Returns

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

2.4.2 H5Dnative_get_section_types

Queries the types of sections for a native dataset.

Signature

```
herr_t H5Dnative_get_section_types (hid_t dset_id,
                                    H5D_native_section_type_t sect_types[])
```

Parameters

dset_id	IN: Dataset ID
sect_types	OUT: Array of (number of sections) of types of sections used to store data for this dataset

Description

H5Dnative_get_section_types() returns the types of sections used to store element values for a dataset with the native VOL connector. Legacy datasets have one section type (H5D_NATIVE_SECTION_FIXED), but other types are possible for sparse datasets and datasets with variable-length datatype storage.

Advice to users

This routine is used to determine the types of sections for calls to H5Dnative_get_chunk_info and H5Dnative_read_chunk.

Returns

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

2.4.3 H5Dnative_write_chunk

Writes pre-processed chunk to a dataset.

Signature

```
herr_t H5Dnative_write_chunk (hid_t dset_id,
                              hid_t dxpl_id,
                              const hsize_t *offset,
                              unsigned num_sections,
                              const H5D_native_chunk_section_info_t section_info[],
                              const void *buf[])
```

Parameters

dset_id	IN: Dataset ID
dxpl_id	IN: Dataset transfer property list ID
offset	IN: Coordinate location of the chunk in the dataset
num_sections	IN: Number of chunk sections provided
section_info	IN: Array of information for each section written
buf	IN: Array of buffers for sections provided

Description

H5Dnative_write_chunk() writes out pre-processed chunk information directly to a dataset. Applications should query for the number and types of sections for a dataset.

Advice to implementors

This routine is backwardly compatible with the existing H5Dwrite_chunk and is designed to be a superset that can also handle structured chunks.

Returns

Returns a non-negative value if successful; otherwise, returns a negative value. Calling this routine on a dataset that is not chunked is an error.

2.4.4 H5Dnative_read_chunk

[[Same as H5Dnative_write_chunk, with appropriate changes for reading vs. writing]]

2.4.5 H5Dnative_get_chunk_info[17]

Queries information about a specific chunk.

Signature

```
herr_t H5Dnative_get_chunk_info (hid_t dset_id,
                                 const hsize_t *offset,
                                 H5D_native_section_info_t section_info[],
                                 haddr_t *chunk_addr[18],
                                 hsize_t *chunk_size)
```

Parameters

dset_id	IN: Dataset ID
offset	IN: Coordinate location of the chunk in the dataset
section_info	OUT: Array of information for each section in the chunk, unmodified if the chunk doesn't exist
chunk_addr	OUT: Address of the chunk in the file, HADDR_UNDEF if the chunk doesn't exist
chunk_size	OUT: Size of the chunk in the file, 0 if the chunk doesn't exist

[17] I suggest just one API routine to query chunk info and to use coordinates for it, instead of the complex (existing) H5Dget_chunk_info and the by-coordinate H5Dget_chunk_info_by_coord.

[18] It's definitely possible that each section could be stored in different locations, instead of as a single block in the file. We may want this routine to be "haddr_t sect_addr[]" instead, with a "real" address for the first section and HADDR_UNDEF for the others when a chunk is stored in a single block. And the same comment for 'chunk_size'.

Description

H5Dnative_get_chunk_info() queries information about the sections for a particular chunk of a dataset. The section_info array provided should have enough room to store the number of sections queried from H5Dnative_get_section_count(). Legacy chunked datasets will always have a section type of H5D_NATIVE_SECTION_FIXED and the section size will be the same as the chunk_size.

Advice to implementors

This routine is backwardly compatible with the existing H5Dget_chunk_info / H5Dget_chunk_info_by_coord and is designed to be a superset that can also handle structured chunks.

Returns

Returns a non-negative value if successful; otherwise, returns a negative value. Calling this routine on a dataset that is not chunked is an error.

2.4.6 H5Dnative_chunk_iter

Iterates over chunks in native storage.

Signature

```
herr_t H5Dnative_chunk_iter (hid_t dset_id,
                             hid_t dxpl_id,
                             H5D_native_chunk_iter_op_t op,
                             void *op_data)
```

Parameters

dset_id	IN: Dataset ID
dxpl_id	IN: Dataset transfer property list ID
op	IN: Callback routine, called for each chunk
op_data	IN: User-defined data, passed to each callback invocation

Description

H5Dnative_chunk_iter() iterates over the chunks for a dataset, invoking the user's op callback for each. The op callback is not invoked for chunks that do not have stored values.

The callback is defined as:

typedef int (*H5D_native_chunk_iter_op_t) (const hsize_t *offset, void *op_data);

Parameters

`offset`	IN: Coordinate of chunk in the dataset
`op_data`	IN: User-defined data, from call to H5Dnative_chunk_iter

The return value from the op callback should be H5_ITER_CONT, H5_ITER_STOP, or H5_ITER_ERROR.

Advice to users

H5Dnative_get_chunk_info, H5Dnative_write_chunk, or H5Dnative_read_chunk can be called from this routine, if the dataset ID is provided as part of the information pointed to by op_data.

Advice to implementors

This routine is a cleaned-up form of H5Dchunk_iter, with metadata parameters that the user may not want / need removed.

Returns

Returns a non-negative value if successful; otherwise, returns a negative value. Calling this routine on a dataset that is not chunked is an error.

2.4.7 H5Dnative_get_chunk_storage_size

[[Same as H5Dget_chunk_storage_size, working for both legacy and structured chunks]]

2.4.8 H5Dnative_get_num_chunks

[[Same as H5Dget_num_chunks, working for both legacy and structured chunks]]

2.5 C Code Examples[19]

This section contains links to some examples. It will be updated when the programming model and APIs are finalized.

`H5Dget_defined` example:
https://gamma.hdfgroup.org/ftp/pub/outgoing/vchoi/SPARSE/hdf5lib_docs/html/group___h5_d.html#title21

[19] This section will be updated after programming model and APIs are finalized.

`H5Derase` example:
https://gamma.hdfgroup.org/ftp/pub/outgoing/vchoi/SPARSE/hdf5lib_docs/html/group___h5_d.html#title11

`H5Dwrite_struck_chunk` example:
https://gamma.hdfgroup.org/ftp/pub/outgoing/vchoi/SPARSE/hdf5lib_docs/html/group___h5_d.html#title46

3 Sparse Matrices Optimized Memory Formats and HDF5 Sparse Storage

There were many requests to provide a mechanism for storing sparse matrices in HDF5 that are represented in memory using different memory optimized formats. Those formats support efficient modifications (e.g., matrices constructions) or efficient access and matrix operations. One can easily store data structures used for sparse matrix representation in memory directly into HDF5. Unfortunately, this creates data portability problems since one has to know the used storage schema and be able to find corresponding data in HDF5 file. The proposed sparse chunk storage is agnostic to the memory structures used to represent sparse matrices. Sparse chunk storage provides storage savings and portability between different memory formats. Of course, such portability does require an extra work as described next.

To store a sparse matrix that uses optimized memory format in HDF5 sparse chunk format, one would need to create HDF5 selection, which describes defined elements, provide a corresponding data buffer, and then pass these to the H5Dwrite call. On the read one would need to convert data buffer and selection, which describes defined elements, into the memory format. This can be a tedious process even for the simple memory schemas as Compressed Sparse Columns (CSC), used by SciPy and MATLAB, or Compressed Sparse Rows (CSR), and, therefore, it would be desirable to provide a high-level library that can do mapping between memory optimized formats to HDF5 sparse storage. This library is outside the scope of the current project. We intend to provide some examples how it can be done and engage the HDF5 community in contributing to such library.

4 Final Recommendation for Supporting Sparse Data in HDF5

To be added after discussions with the HDF5 community.

5 Appendix

The table below lists new recommended functions for working with sparse dataset or datasets that use structured chunk storage.

Function Name	Short Description	Document Section Number
H5Pset_struct_chunk	Sets structured chunked storage	2.1.1
~~H5Pset_sparse_chunk~~	Sets sparse chunked storage	2.1.3
H5Dget_defined	Retrieves a dataspace object with the defined elements	0
H5Derase	Deletes elements from a dataset	2.1.5
H5Dwrite_struct_chunk[20]	Writes structured chunk	2.2.1
H5Dread_struct_chunk	Reads structured chunk	2.2.2
H5Dget_struct_chunk_info	Gets structured chunk info	2.2.3
H5Dget_struct_chunk_info_by_coord	Retrieves the structured chunk information	2.2.4
H5Dstruct_chunk_iter	Iterates over all structured chunks in the dataset	2.2.5
H5Pset_filter2	Adds a filter to a filter pipeline for a specified section of sparse structured chunk	2.3.1
H5Pget_nfilter2	Returns the number of filters in the pipeline for a section of structured chunk	2.3.1
H5Pget_filter3	Returns information for a filter in the pipeline for a specified section	2.3.1
H5Pget_filter_by_id3	Returns information for a filter specified by its identifier in the pipeline for a specified section of structured chunk	2.3.1
H5Premove_filter2	Removes a filter in the filter pipeline for a specified section	2.3.1
H5Pmodify_filter2	Modifies a filter in the filter pipeline for a specified section of structured chunk	2.3.1

[20] Our recommendation is to version existing proposed 5 direct chunk functions using provided signatures.

Acknowledgment

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

The authors would like to thank The HDF Group developers and Quincey Koziol, Principal Engineer, AWS HPC for reviewing the numerous versions of the document and for fruitful discussions.

References

1. The HDF Group, Draft RFC: Sparse Chunks, https://docs.hdfgroup.org/hdf5/rfc/RFC_Sparse_Chunks180830.pdf
2. J. Mainzer *et al*., "Sparse Data Management in HDF5," *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*, Denver, CO, USA, 2019, pp. 20-25, doi: 10.1109/XLOOP49562.2019.00009.
3. The HDF Group, “HDF5 File Format Specification” https://www.hdfgroup.org/HDF5/doc/H5.format.html
4. John Mainzer, Elena Pourmal, “RFC: File Format Changes for Enabling Sparse Storage in HDF5”. Available from https://github.com/LifeboatLLC/SparseHDF5/
5. The HDF Group, Variable-Length Data in HDF5 Sketch Design, https://docs.hdfgroup.org/hdf5/rfc/var_len_data_sketch_design_190715.pdf
6. The HDF Group, API Compatibility Macros, https://docs.hdfgroup.org/hdf5/develop/api-compat-macros.html

Revision History

March 17-May 1, 2023:	Version 1-5 were created for internal reviews.
May 2, 2023	Version 6 was sent to THG for review.
May-June 2023	Versions 7-8 were created for internal review. The document was updated according to the changes in the File Format RFC and feedback from THG.
June 14, 2023	Version 9 was sent to THG for review.
June 14-23, 2023	Version 10-12 added functions to work with structured chunk sections.
June 26, 2023	Version 13 is prepared for review by THG.
July 10, 2023	Version 14 was prepared for THG review. We introduced new versions of the existing filter APIs instead of adding new functions.
July 18, 2023	Version 15 was prepared for public review. Fixed typos.
August 3, 2024	Version 16-17 – added new functions H5Pset_structured(sparse)_chunk; cleaned comments
August 14, 2024	Version 18 - Synced with Doxygen and added links to Doxygen docs available at https://gamma.hdfgroup.org/ftp/pub/outgoing/vchoi/SPARSE/hdf5lib_docs/html/_r_m.html
October 23, 2024	Version 19 – Removed Tools section since there is now Tools RFC "RFC: Changing Tools to Support Sparse Data in HDF5".
October 24, 2024	Version 20 – Public comments and suggestion were added and addressed. Lifeboat and THG developers will revisit APIs and programming model after the first prototype implementation of sparse storage and shared chunk cache is completed.
April 22, 2025	Version 21 – copy editing of Version 20, added foot notes with recommendations for `chunk_info` data structure and API names for direct chunk operations.