RFC: Programming Model to Support Sparse Data in HDF5

John Mainzer (john.mainzer@lifeboat.llc),
Elena Pourmal (elena.pourmal@lifeboat.llc)
Lifeboat, LLC

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. Section 4 outlines necessary changes to the command-line tools.

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.

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[1] 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

[1] 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.

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 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][2] 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 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.

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

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”[3] 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 `H5Pset_layout` and **new storage layout** `H5D_SPARSE_CHUNK`.
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.
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

[3] 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.

"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[4]

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

2.1.1 H5Dget_defined

Retrieves a 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 a negative value.

2.1.2 H5Derase

Deletes elements from a dataset.

Signature

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

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

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 layout `H5D_SPARSE_CHUNK`. For other layouts this function will return an error.[5]

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.

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

[5] 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.

```
typedef struct H5D_struct_chunk_info_t {
    enum        type;                /* Type of the structured chunk;          */
                                     /* 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
```

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: one for the encoded selection and the second one for data elements.

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.

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 chunk has only two sections: one for the encoded selection and the second one for data 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.

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.

Signature

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

Parameters

<table>
<tr><td>plist_id</td><td>IN: Object creation property list identifier</td></tr>
<tr><td>section_number</td><td>IN: An integer to specify section number. The value is 0 to 255 when native HDF5 file format is used.

For the sparse chunk the convenience flag can be used to specify a section of the sparse chunk to be filtered as described below
<table>
<tr><td>H5Z_FLAG_SPARSE_SELECTION</td><td>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.</td></tr>
<tr><td>H5Z_FLAG_SPARSE_FIXED_DATA</td><td>Adds the filter to the filter pipeline for section 1 of the sparse chunk. It has the same effect as passing 1.</td></tr>
<tr><td>H5Z_FLAG_SPARSE_VL_DATA</td><td>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.</td></tr>
</table></td></tr>
<tr><td>filter</td><td>IN: Filter identifier for the filter to be added to the pipeline</td></tr>
<tr><td>flags</td><td>IN: Bit vector specifying certain general properties of the filter</td></tr>
<tr><td>buf_size</td><td>IN: Size in bytes of buf buffer</td></tr>
<tr><td>buf</td><td>IN: Buffer with an auxiliary data for the filter</td></tr>
</table>

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.

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.

```
int H5Pget_nfilter2 (hid_t plist_id, uint64_t section_number);
```

/* Returns the number of filters in the pipeline for a section of structured chunk */

```
H5Z_filter_t H5Pget_filter2 (hid_t dcpl, uint64_t section_number, unsigned idx,
uint64_t *flags, size_t *buf_size, void *buf, size_t namelen, char name[],
unsigned *filter_config);
```

/* Returns information for a filter in the pipeline for a specified section */

```
H5Z_filter_t H5Pget_filter_by_id (hid_t dcpl, uint64_t section_number,
H5Z_filter_t filter, 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 */

```
herr_t H5Premove_filter2 (hid_t plist_id, uint64_t section_number, H5Z_filter_t
filter);
```

/* Removes a filter in the filter pipeline for a specified section */

```
herr_t H5Pmodify_filter2 (hid_t plist_id, uint64_t section_number, H5Z_filter_t
filter, uint64_t flags, size_t buf_size, const void *buf);
```

/* Modifies a filter in the filter pipeline for a specified section of structured chunk */

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;
status = H5Pset_filter2 (dcpl, sec_num, H5Z_FILTER_DEFALTE, flags, …);
sec_num = H5Z_FLAG_SPARSE_FIXED_DATA;
status = H5Pset_filter2 (dcpl, sec_num, H5Z_FILTER_DEFALTE, flags, …);
```

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_DEFALTE, 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.

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 C Code Examples

2.4.1 Example 1 Setting sparse storage and compression

In this example we show how to set sparse chunk storage using new API H5Pset_layout and existing functions to set up compression on two sections of the sparse chunk (serialized selection section and the fixed-size section that holds defined elements of integer type).

```
….
    H5Z_filter_t    filter_type;
    hsize_t         chunk_dims[2] = {CHUNK0, CHUNK1};
    uint64_t        flags;
    size_t          nelmts = 1;
….. unsigned int    data = 9;

    /*
     * Create the dataset creation property list, add the gzip
     * filter to compress data elements of sparse array and set the chunk size.
     */
    dcpl = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_layout (dcpl, H5D_SPARSE_CHUNK);
    status = H5Pset_chunk (dcpl, 2, chunk_dims);

    /* Apply compression methods to different sections of
     * a structured chunk. In this example, sparse chunk has two sections.
     * We are using gzip compression on the encoded selection section
     * and szip on the fixed-size data section.
     */
    flags = H5Z_FLAG_OPTIONAL;

    status = H5Pset_filter2 (dcpl, H5Z_FLAG_SPARSE_SELECTION,
                               H5Z_FILTER_DEFALTE, flags, nelem, &data);

    status = H5Pset_filter2 (dcpl, H5Z_FLAG_SPARSE_FIXED_DATA,
                               H5Z_FILTER_SZIP, flags, …);

    /*
     * Create the dataset.
     */
    dset = H5Dcreate (file, DATASET, H5T_STD_I32LE, space, H5P_DEFAULT, dcpl,
                H5P_DEFAULT);

    /*
     * Write the data to the dataset.
     */
    status = H5Dwrite (dset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                wdata[0]);
```

To find information about the applied filters one can use H5Pget_nfilters2 to find the number of applied filters and then use H5Pget_filter2 to find information about filters used on specific section. Example below assumes that Encoded Selection section of the structured chunk was filtered.

```
    nelmts = 0;
    idx = 0;
    filter_type = H5Pget_filter2 (dcpl, H5Z_FLAG_SPARSE_SELECTION, idx,
                                    &flags, &nelmts, NULL, 0, NULL,
                                    &filter_info);
    switch (filter_type) {
        case H5Z_FILTER_DEFLATE:
        printf ("H5Z_FILTER_DEFLATE is applied to selection section\n");
        break;
    }
```

2.4.2 Example 2 Using predefined filter function H5Pset_deflate

We can use predefined filter functions on all sections of the sparse chunk.

```
    /*
     * Create the dataset creation property list, add the gzip
     * filter to compress all sections of the sparse chunk using DEFLATE filter.
     */
    dcpl = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_layout (dcpl, H5D_SPARSE_CHUNK);
    status = H5Pset_chunk (dcpl, 2, chunk_dims);
    status = H5Pset_deflate (dcpl, 9);
    /*
     * Create the dataset.
     */
    dset = H5Dcreate (file, DATASET, H5T_STD_I32LE, space, H5P_DEFAULT, dcpl,
                H5P_DEFAULT);

    /*
     * Write the data to the dataset.
     */
    status = H5Dwrite (dset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                wdata[0]);
```

As one can see from the two examples above, there is no any substantial changes to the programming model when working with the sparse data.

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 Changes to the Command-Line Tools

In general, all command-line tools should work with sparse data as they work with the dense data. Of course, this statement requires confirmation. Existing tools tests and files have to be updated to include sparse datasets.

In this RFC we outline the changes to a few tools that would require changes to the user's interface, i.e., additional flags to display locations of the defined elements or to the output to specify the usage of sparse chunk storage. Here we only outline the major changes and will leave the exact specifications to another tools specific RFC.

4.1 h5dump

DDL used by h5dump output should be updated to indicate new storage type and locations of the "defined" data elements.

Here we provide an example how h5dump output may look like for a dataset created in Example 1 that uses sparse storage. "SPARSE CHUNK" keyword indicates new storage type. The rest of the output is similar to the output for a dataset with chunked storage and GZIP compression applied to each section of the structured chunk.

```
$ hdf5/bin/h5dump -p -H h5sparse.h5
HDF5 "h5sparse.h5" {
GROUP "/" {
   DATASET "DS1" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 32, 64 ) / ( 32, 64 ) }
      STORAGE_LAYOUT {
         SPARSE CHUNK ( 5, 9 )
         SECTION 0 <section_name>[6] SIZE 100 (2:1 COMPRESSION)
         SECTION 1 <section_name> SIZE 5018 (1.633:1 COMPRESSION)
      }
      FILTERS SECTION 0 {
         COMPRESSION DEFLATE { LEVEL 9 }
      }
      FILTERS SECTION 1 {
         COMPRESSION SZIP {
            PIXELS_PER_BLOCK 4
            MODE K13
            CODING NEAREST NEIGHBOUR
            BYTE_ORDER LSB
```

[6] We may provide sections names that are symbols used to set compression flag, e.g., H5Z_FLAG_SPARSE_SELECTION

```
            HEADER RAW
         }
      }

      FILLVALUE {
         FILL_TIME H5D_FILL_TIME_IFSET
         VALUE  0
      }
      ALLOCATION_TIME {
         H5D_ALLOC_TIME_INCR
      }
   }
}
}
```

We should note here that existing sub-setting flags can be used to specify bounding box to print sparse data as usual using fill values for data that is not defined. It would be helpful to introduce new flag to print locations of the defined elements, for example, `--sparse-locations`, to print the locations of the defined elements. The output will contain the coordinates of the individual elements or the coordinates of the "upper left" and "low right" simple hyperslab if all points are defined in that hyperslab. Such formats are already used to print hyperslab and point selections (see the format used to print `RIGION_TYPE BLOCK`, e.g. `(0,0) – (8,8)` will represent a subarray of the size 9x9 located in the left upper corner of a matrix, and the format used to print `RIGION_TYPE POINT`[7], e.g., `(0,0),(1,1), …,(N,N)` will represent a diagonal elements of `N x N` matrix. For example, for the matrix show by

Figure 3 stored as a dataset "Sparse" using sparse chunk layout, the output for data locations may look like as shown here:

```
DATASET “Sparse” {
            DATATYPE  H5T_STD_U8BE
            DATASPACE  SIMPLE { ( 13, 10 ) / ( 13, 10 ) }
            REGION_TYPE BLOCK (2,2)-(4,7)
            REGION_TYPE BLOCK (6,0)-(6,2)
            REGION_TYPE POINT (5,9), (11, 1), (12,8)

            }
         }
```

[7] Initial implementation will work with the hyperslab selections only. We use this as an example if in the future implementation will be extended to allow the point selections. Currently one will need to use hyperslab selection to specify one element, i.e., the line for the individual points `REGION_TYPE POINT (5,9), (11, 1), (12,8)` in h5dump output above will become
REGION_TYPE BLOCK (5,9) - (5,9)
REGION_TYPE BLOCK (11,1) - (11,1)
REGION_TYPE BLOCK (12,8) - (12,8)

Figure 3: “Sparse” dataset shown by h5dump output above

Row/column index	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	
2	0	0	66	69	72	75	78	81	0	0
3	0	0	96	99	102	105	108	111	0	0
4	0	0	126	129	132	135	138	141	0	0
5	0	0	0	0	0	0	0	0	0	2
6	100	0	-100	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	1	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	3	0

It is also desired to print both locations and data at those locations. We will need another flag, e.g., --sparse, that may provide an output shown below:

```
   DATASET “Sparse" {
           DATATYPE  H5T_STD_U8BE
           DATASPACE  SIMPLE { ( 13, 10 ) / ( 13, 10 ) }
           REGION_TYPE BLOCK  (2,2)-(4,7)
           DATA {
           (2,2)   66, 69, 72, 75, 78, 81,
           (3,2)   96, 99, 102, 105, 108, 111,
           (4,2)   126, 129, 132, 135, 138, 141
           }
           REGION_TYPE BLOCK (6,0)-(6,2)
           DATA {
           (6,0)   100, 0, -100
           }
           REGION_TYPE POINT (5,9), (11, 1), (12,8)
           DATA {
           (11,1) 1
           (5,9) 2
           (12,8) 3
           }
        }
```

We will print the coordinates of the first element on each line of the output. In the example above two BLOCK segments represent simple hyperslabs and their data, and the POINT segment shows the list of all point selections (e.g., coordinates of the elements and the values).

4.2 h5ls

Since h5ls doesn’t have flags for specifying sub-setting and de-referencing region references and doesn’t use corresponding output format as h5dump, we suggest no changes to the tool (i.e., any new flags) to print defined sparse data elements. The only required change will be to indicate new type of storage “Sparse Chunks” (vs. current “Chunks”) as shown below:

```
% h5ls -rv tfilters.h5
Opened "tfilters.h5" with sec2 driver.
/                        Group
    Location:  1:96
    Links:     1
/all                     Dataset {20/20, 10/10}
    Location:  1:29336
    Links:     1
    Sparse Chunks: {10, 5} 200 bytes
    Section 1 <section_name>
    Storage:   800 logical bytes, 458 allocated bytes, 174.67% utilization
    Filter-0:  shuffle-2 OPT {4}
    Filter-1:  szip-4 OPT {141, 4, 32, 5}
    Filter-2:  deflate-1 OPT {5}
    Filter-3:  fletcher32-3  {}
    Filter-4:  nbit-5 OPT {8, 1, 50, 1, 4, 0, 32, 0}
    Type:      native int
    Section 2 <section name>
    ……
```

4.3 h5stat

We will need to updated the tool’s output to display the number of the datasets with sparse chunk layout. For example, if a file contains three sparse datasets along with the datasets with other storage layouts, the output will look like this:

```
Dataset layout information:
     Dataset layout counts[COMPACT]: …
     Dataset layout counts[CONTIG]: …
     Dataset layout counts[CHUNKED]: …
     Dataset layout counts[STRUCTURED CHUNK SPARSE]: 3
     Dataset layout counts[VIRTUAL]: …
     …..
```

4.4 h5import

The biggest challenge for the tool will be to define new input class for sparse data provided as an ASCII file or binary file. We should take into consideration common sparse data storage schemas in both ASCII and binary files (see discussion in section 3). Currently, this is out of scope of this project.

Another observation is since the tool is interoperable with h5dump, we will need to preserve this option for sparse data when consider updates to configuration file and input data file. I.e., it would be beneficial to co-design h5dump and h5import changes.

4.5 h5diff

The tool doesn't compare layouts when diffing two datasets, therefore, no changes to the flags or tool's output will be required.

4.6 h5repack

h5repack should be updated to repack a dataset from/to sparse chunk storage layout.

Existing flags can be used to change storage from sparse chunk to the chunked and to the contiguous storage layouts.

We will need to introduce a new flag, e.g., "SPARSECHUNK", when repacking to use sparse chunk storage. Please notice that without specifying defined values sparse chunk storage will not achieve space savings since all data elements of each chunk will be stored along with the dataspace information for the chunk. In order to take advantage of sparse chunk storage we will need to introduce new flag, for example, `--defined-elements`, followed by the list of BLOCKs and POINTs using format as specified by the h5dump tool's DDL as shown next. We assume that example matrix shown on

Figure 3 is stored as a regular HDF5 dataset and is repacked to use sparse chunk storage layout using new option and known locations for defined elements.

```
% h5repack -l Sparse:SPARSECHUNK=5x4 BLOCK (2,2)-(4,7), (6,0)-(6,2) POINT
(5,9), (11, 1), (12,8) dense.h5 sparse.h5
```

This approach may not work for huge datasets and some automation for creating BLOCK/POINT input list or automated detection of "defined value" would be required. For example, h5dump can be updated to print locations in the required BLOCK/POINT format only for the values that are not equal to some specified value, or h5repack may scan the data and exclude the elements that have a specified value. For example,

```
% h5repack -l Sparse:SPARSECHUNK=5x4 –exclude 0 dense.h5 sparse.h5
```

Such approach is problematic as shown by the following example. In our original example of sparse matrix stored using sparse chunk layout, the second element in the `BLOCK (6,0) – (6,2)` has value 0 and it is defined. If we repack the file to dense layout the default library fill value 0 will be used for undefined values. If we repack back to the sparse chunk layout using "`–exclude 0`", the file will be different since the original `BLOCK (6,0)-(6,2)` will become `POINT(6,0), (6,2),` i.e. we will violate round trip behavior for h5repack.

5 Final Recommendation for Supporting Sparse Data in HDF5

To be added after discussions with the HDF5 community.

6 Appendix

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

Function Name	Short Description	Document Section Number
`H5Dget_defined`	Retrieves a dataspace object with the defined elements	2.1.1
`H5Derase`	Deletes elements from a dataset	2.1.2
`H5Dwrite_struct_chunk`	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	2.3.2
`H5Pget_filter2`	Returns information for a filter in the pipeline for a specified section	2.3.2
`H5Pget_filter_by_id2`	Returns information for a filter specified by its identifier in the pipeline for a specified section of structured chunk	2.3.2
`H5Premove_filter2`	Removes a filter in the filter pipeline for a specified section	2.3.2
`H5Pmodify_filter2`	Modifies a filter in the filter pipeline for a specified section of structured chunk	2.3.2

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.