Structured Storage for Sparse and Variable-Length Data

Benchmark results for storage prototype

Elena Pourmal (elena.pourmal@lifeboat.llc)
Raymond Lu (raymond.lu@lifeboat.llc)
Lifeboat, LLC

We created benchmarks to prototype the structured chunk storage for sparse and variable-length data in HDF5. Structured chunk storage for these kinds of data is discussed in [1, 2]. This document describes our benchmarking experiments to support proposed new storage design and summarizes the findings.

The benchmarks source code and results are available from https://github.com/LifeboatLLC/SparseHDF5/tree/main/benchmarks.

1 Introduction

Lifeboat, LLC has been working on designing sparse data storage in HDF5. In this document we provide benchmarking results to support our design decisions.

In HDF5, problem-sized data is stored in multidimensional arrays of elements of a given type. Currently, the HDF5 library requires that *all* elements are defined with user-supplied values or fill values, i.e., it treats data as "dense", mapping *each* data element to storage during I/O operations. HDF5 chunking and per-dataset compression help to optimize the storage of sparse data. However, there are several obvious disadvantages to applying "dense storage" to sparse data: the location of the user-supplied data is not explicitly represented; when read into memory (and after decompression), it may result in a huge memory footprint. Therefore, a different approach for sparse data management was needed. We introduced the notion of structured chunk to store sparse data in HDF5 and outlined the required extensions to the HDF5 file format in [1] and public APIs in [2] to support this new storage paradigm. We also showed that it can be used for storage of variable-length data. To justify our design decisions, we prototyped structured chunk storage with the current HDF5 capabilities as described in sections 0 and 2 of this document.

To set up a context for the benchmarks and discussions of the results, we would like to outline the idea of the structured chunk storage using examples of sparse numeric data and dense variable-length data. For details on structured chunk storage, we refer the reader to the documents cited above.

Like a regular chunk, a structured chunk stores the values contained in some n-dimensional rectangular volume in a dataset. However, unlike regular chunk, it is composed of two or more sections, which in combination, describe the values contained in the volume. For example, the structured chunk that contains sparse data[1] of fixed-size datatype will have two sections: one section to store encoded coordinates (i.e., hyperslab selection) of the defined elements and another section to store the values of the defined elements. When storing data of HDF5 variable-length datatype, the structured chunk will also have two sections: a section (a heap) with the variable-length elements (e.g., strings), and a section with the pointers to the elements (e.g., offsets into the heap and lengths of the strings). When storing sparse data of *variable-length datatype* the structured chunk will have *three* sections: a section to store encoded coordinates of the defined elements, a section with the defined variable-length elements, and a section with the pointers to the defined elements.

Our design supports data filtering that can be applied to different sections of the structured chunk, for example, when storing sparse data, the section containing encoded selection and the section containing values of defined elements can be compressed independently using different compression methods.

The goal of the work summarized in this document was to evaluate the new storage approach[2]. In our benchmarks we store sparse and variable-length data in HDF5 datasets as it is done now and in a way that emulates structured chunk with the sections as described in the examples above. Each section of

[1] In a sparse data array only some elements are "defined", i.e., supplied by a user. It is common to represent non-defined elements with some reserved value, for example, 0, when sparse data is stored in an HDF5 dataset.
[2] We would like to emphasize that we focused on prototyping storage properties and not on measuring I/O performance.

the structured chunk is stored as a one-dimensional dataset that has only one chunk, with and without compression allowing us to estimate the total size of the structured chunk and to compare it with the size of the storage used for the current approach. We also looked into the impact of “compression by section” on the structured chunk storage. We should note here that structured chunk metadata is a few bytes bigger than metadata for a regular chunk. At this point we ignore metadata sizes since our focus is on *raw data storage* only.

In the next sections we describe the benchmarks and our experiments, and provide the results for emulated structured chunk storage[3] vs. current HDF5 storage capabilities.

2 Structured Chunk Storage for Sparse Data

This section describes the benchmark, experimental set up and results when using structured chunk approach for storing sparse data. We compare the current way of storing sparse data with the proposed structured chunk storage.

2.1 Sparse benchmark setup

The program `sparse.c` generates an HDF5 file `sparse_file.h5` with a sparse 2-dim data array stored under the group `"percent_X"`, where `X=1,…,M` indicates percentage of the defined values, using two options: as a regular 2-dim HDF5 dataset `"sparse"` in which undefined elements are represented by `0` or as two 1-dim HDF5 datasets. In the latter case, the dataset `"selection"` contains the encoded hyperslab selection of the defined elements in the data array and the dataset `"data"` contains defined elements themselves. These two 1-dim datasets emulate two corresponding sections of the structured chunk proposed for storing sparse data of the fixed-size datatype. The group `"percent_X"`, also contains compressed versions of the three datasets described above. The value of `M` is specified with a command line option `-m`.

All datasets use chunked storage with a rank and dimension sizes of the chunk being equal to the rank and dimension sizes of the corresponding dataset, i.e., each dataset has only one chunk. The chunk sizes for the 2-dim sparse data array are specified with a command line option `-c` as multiples of 1024 bytes. The sizes of the 1-dim HDF5 datasets depend on the values of `X` and a type of the hyperslab selection and are determined after the data and the hyperslab selection have been generated.

The program generates three types of hyperslab selection based on a value (`1`, `2`, or `3`) provided with a command line option `-s`:

- random locations in each row (`1`)
- randomly placed rectangular sub-region in the entire chunk (`2`)[4]
- randomly placed continuous locations in each row (`3`)

[3] Provided results for emulated structured chunk storage are for the sparse data with the fixed-size datatype and for the variable-length data with the fixed-size base datatype. More complex datatypes for sparse and variable-length data were out of scope for this study.
[4] For simplicity of the code, we restricted the position of the rectangular selection by placing its upper-left corner always in the upper-left quadruple of the chunk.

The values of the defined elements of the sparse array are generated based on a value (`0` or `1`) provided with a command line option -d. The program will generate and initialize data with

- the sequences `1,2,...,N,` where `N =< UCHAR_MAX (0)`
- the random values between 1 and `UCHAR_MAX (1)`

`h5cc` can be used to compile the program.
Example: The commands

```
     h5cc sparse.c
    ./a.out -c 1x1 -m 3 -s 2
```

will generate `sparse_file.h5` with the following file structure shown by the `h5dump -n` command:

```
HDF5 "sparse_file.h5" {
FILE_CONTENTS {
 group /
 group /percent_1
     dataset /percent_1/data
     dataset /percent_1/data_comp
     dataset /percent_1/selection
     dataset /percent_1/selection_comp
     dataset /percent_1/sparse
     dataset /percent_1/sparse_com
 group /percent_2
     dataset /percent_2/data
     dataset /percent_2/data_comp
     dataset /percent_2/selection
     dataset /percent_2/selection_comp
     dataset /percent_2/sparse
     dataset /percent_2/sparse_comp
 group /percent_3
     dataset /percent_3/data
     dataset /percent_3/data_comp
     dataset /percent_3/selection
     dataset /percent_3/selection_comp
     dataset /percent_3/sparse
     dataset /percent_3/sparse_comp
 }
}
```

Datasets "sparse" and its compressed counterpart "`sparse_comp`" will have dimensions 1024x1024 with data density varying from 1 to 3%. All defined elements are located in a randomly placed rectangular sub-region (hyperslab) of the dataset.

We ran the program with `m=10` for the three types of hyperslab selection described above generating random or compressible raw data. Our goal was to evaluate storage requirements for the structured chunk with and without data compression and compare it with the current HDF5 storage for sparse data.

In the tables below we provide the results only for the datasets stored in `/percent_10` group and chunk size `1024x1024` since they are similar for other percentage and chunk sizes. Comprehensive results are provided in Appendix 1. The storage savings trends for different data densities (percentage) are shown on the graphs.

2.2 Compression and hyperslab selection encoding

Table 1 shows sizes of encoded selection for each selection type, its compressed counterpart and corresponding compression ratio. In the table we used data from the experiments with compressible raw data since the hyperslab generation process is the same for the random raw data.

Selection type	Original size	Compressed size	Compression ratio
1	1654231	184324	9
2	87	51	2.0
3	16439	6718	2.4

Table 1: Original and compressed sizes of encoded selection stored in the "`selection`" and "`selection_comp`" datasets with 10% fill and corresponding compression ratio (size of the encoded selection/size of compressed encoded selection) based on the selection type as shown in Table 10, Table 16, and Table 22

Figure 1 shows compression ratio for three types of the hyperslab selection and density (percentage) varying from `1` to `10`.

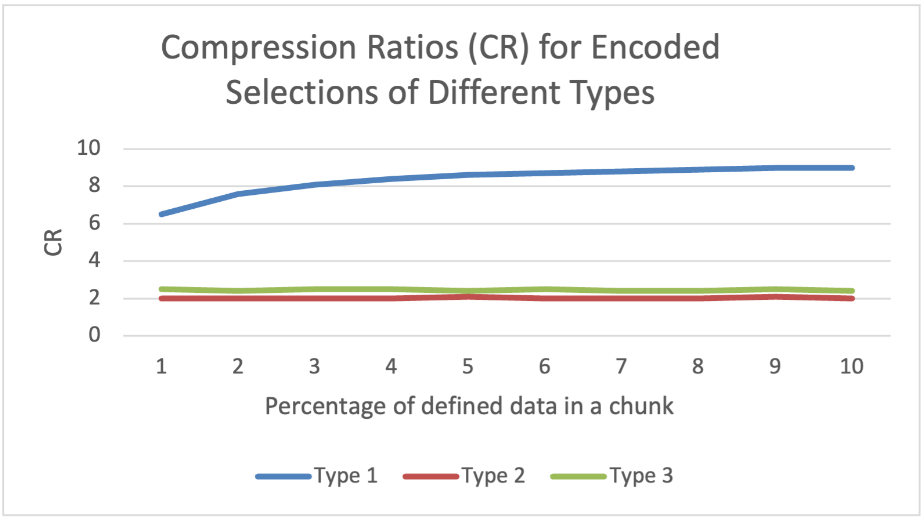

Figure 1: Compression ratios for encoded HDF5 selections for randomly selected locations (type 1), randomly placed rectangle selection (type 2), and randomly placed continuous locations in each row of 2-dim sparse array (type 3).

We see that compression works well on all tested selection types with CR value being greater or equal to 2. Compression is especially beneficial for the selection consisting of the random points (type 1).

Please notice, that currently, HDF5 doesn't have a capability to encode point selections (type 1), therefore, each isolated location is encoded as a hyperslab with length 1 in each dimension. The encoding of an HDF5 hyperslab uses upper left and low right coordinates of the hyperslab that are the same for a selected point thus storing duplicated data. Compression helps to mitigate the issue, but, in general, HDF5 should provide point selection encoding too.

2.3 Chunked vs. structured chunk storage without compression

Table 2 shows the size of the original storage, i.e., the size of "`sparse`", vs. structured chunk storage size which is the sum of the sizes of the "`selection`" dataset and the "`data`" datasets for random data. No compression was used. Storage ratio trends for random data is presented on Figure 2. Since compression is not used the plot is the same for compressible data.

When compression is not used, new structured storage provides obvious storage savings for all types of selections especially for the low density since the overhead of storing locations of the defined elements is much smaller than the size of the sparse array. For random locations selections (type 1) storage ratio is under 1 after density is greater than 6% (see Table 11 and Table 14 in Appendix) due to dominated size of the hyperslab encoding. As Table 3 shows, compression helps to slightly mitigate the issue for this type of selection.

Selection type	Original storage	Structured chunk storage	Storage ratio
1	1048576	1758679	0.6
2	1048576	104416	10
3	1048576	120887	8.7

Table 2: Sizes in bytes for original and structured chunk storage using random data for three types of the hyperslab selection and 10% fill.

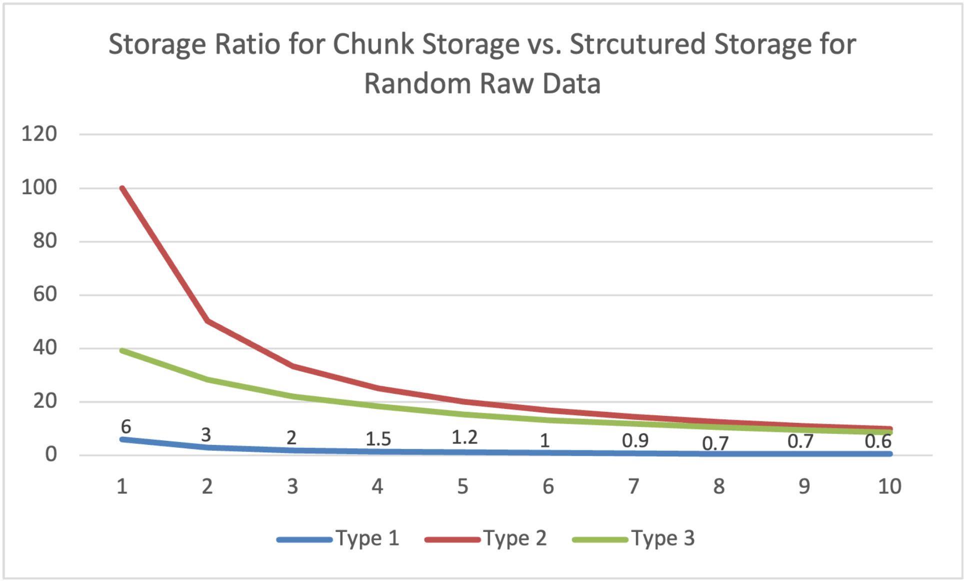

Figure 2: Storage ratio for chunk storage vs. structured storage for three types for hyperslab selection.

2.4 Chunked vs. structured chunk storage using compression

Table 3 shows the size of compressed original storage (i.e., the size of "`sparse_comp`") vs. compressed structured chunk storage size which is the sum of the sizes of the "`selection_comp`" dataset and the "`data_comp`" dataset. In this experiment we used random raw data that is not compressible.

Selection type	Original storage (compressed)	Structured chunk storage (compressed)	Storage ratio
1	209666	288813	0.7
2	108308	104413	1.03
3	11111	111207	0.99

Table 3: Sizes in bytes for original and structured chunk storage using random data and compression for three types of the hyperslab selection.

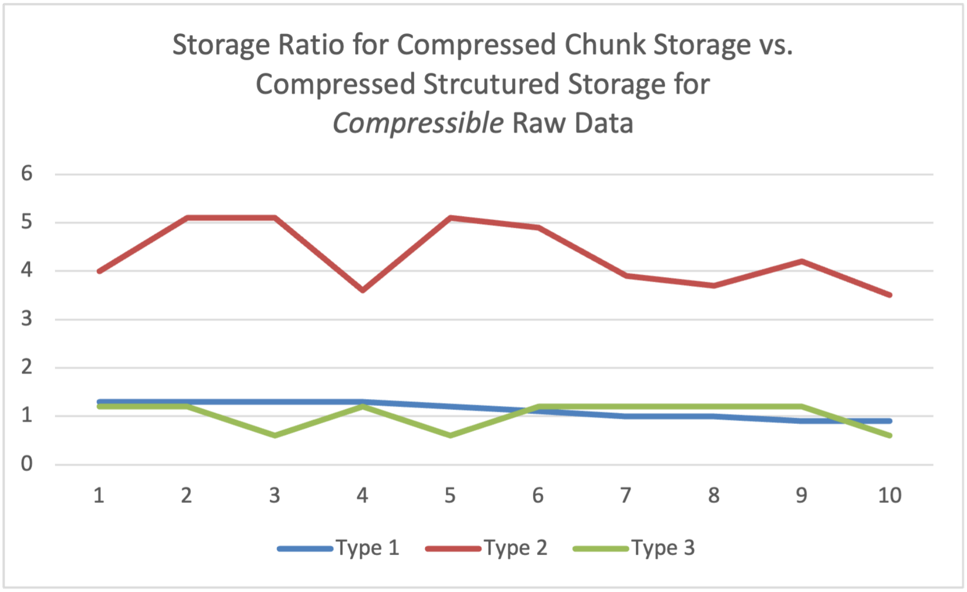

Figure 3: Storage ratio for compressed chunk storage vs. compressed structured chunk storage for different hyperslab selection types using compressible raw data.

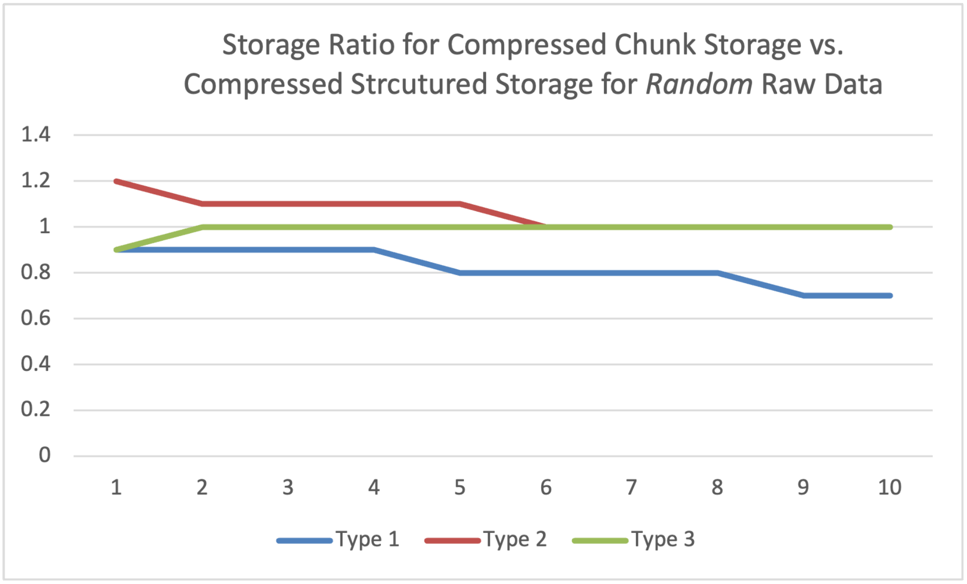

Figure 4: Storage ratio for compressed chunk storage vs. compressed structured chunk storage for different hyperslab selection types using random raw data.

For both cases above we see that structured chunk storage is comparable or provides savings in storage except the case of the random selection (type 1). In this case the size of the encoded selection (1654231 bytes; see
Table 1) is much bigger that the size of the raw data (104448 bytes, derived from the size of structured chunk data in Table 11) and since data is random, compression of the selection and raw data doesn't help. The results are similar when raw data is compressible as shown in

Selection type	Original storage (compressed)	Structured chunk storage (compressed)	Storage ratio
1	165018	185063	0.89
2	2709	780	3.5
3	4126	7478	0.5

Table 4. We have comparable or good storage savings for selection type 1 and 2 but not for type 3. Said this we should remember that with structured chunk storage we store not only the values but their locations too.

Selection type	Original storage (compressed)	Structured chunk storage (compressed)	Storage ratio
1	165018	185063	0.89
2	2709	780	3.5
3	4126	7478	0.5

Table 4: Sizes in bytes for original and structured chunk storage using compressible data and compression for three types of the hyperslab selection.

The results can be summarized as follows:

1. Encoded selection can be successfully compressed; therefore, our design decision to allow compression by section will benefit sparse data storage. See Figure 1.
2. Our main use case for sparse data (selection of type 2) and compressible data showed very good storage savings as shown by the red line on Figure 3.
3. Structured chunk storage for sparse data achieves comparable or better storage size than the current approach along with the provided benefit of stored data locations. See Figure 3 and Figure 4.

3 Structured Chunk Storage for Variable-Length (VL) Data

This section describes the benchmark, experimental set up and results when using structured chunk approach for storing VL data. We compare the current way of storing VL with the proposed structured chunk storage.

3.1 VL benchmark setup

The `vl.c` program generates `N` variable-length (VL) elements of the length `l`, where `l` is a random number between `1` and `M`, and stores them in four different HDF5 files. The table below describes the content of each file.

File name	**File content**

`vltype.h5`	One-dimensional dataset in a `root` group with the elements of HDF5 VL datatype (current way of storing VL data in HDF5)
`vltype_comp.h5`	One-dimensional dataset as above but compressed with GZIP level 9
`vltype_struct.h5`	Two one-dimensional datasets in a `root` group that emulate storage properties of the structured chunk for variable-length data: the first dataset contains pairs of offset/length for each VL element stored in a blob; the second dataset contains the blob with all VL elements. These two datasets represent two sections of the structured chunk.
`vltype_struct_comp.h5`	Two datasets as above but compressed with GZIP deflate level 9

The program has an option to generate VL vectors with `l` random values (we call them "random data") between `0` and 127 or the values `0,1,2...,l` that can be compressed (we call them "compressible data"). All datasets store its data in a single chunk for fair comparison of regular chunk storage with the proposed structured chunk storage.

3.2 VL benchmark experiment and results

We ran the program to generate `10000` VL vectors (random data) of the length between `1` and `10`, and `1` and `100`, and used the `h5stat` tool to find the sizes of metadata and raw data in each file. The results are shown in Table 5 and Table 6 respectively. We also rerun the same settings using compressible data. The results are shown in Table 7 and Table 8 respectively. The columns in the tables have the following meanings:

File size	File size in bytes as reported by `ls` or `h5stat`
SR	Storage size ratio between two files, e.g., (size of `vltype.h5`)/ (size of `vltype_struct.h5`)
Metadata size	HDF5 metadata in bytes as reported by h5stat
Raw data size	User's raw data in bytes as reported by h5stat. In the case of the current storage "raw data size" is the size of an array of pointers to the global heap that stores actual VL elements. In the case of the structured chunk emulation, it is a sum of sizes of two datasets: a dataset with offset/lengths and a dataset with VL elements.
Unaccountable space size	The space in bytes in the global heap used for storing VL elements as reported by `h5stat`.

	File name M=10	**File size**	**SR**	**Metadata size**	**Raw data size**	**Unaccountable space**
1	vltype.h5	425640	1.9	3496	160000	262144
2	vltype_struct.h5	220847		5864	214983	0
3	vltype_comp.h5	290982	4.2	3496	25342	262144
4	vltype_struct_comp.h5	69296		5864	63432	0

Table 5: h5stat statistics for random 10000 VL vectors (random data) with the lengths between 1 and 10 stored in HDF5 files with and without compression.

	File name M=100	**File size**	**SR**	**Metadata size**	**Raw data size**	**Unaccountable space**

1	vltype.h5	884392	1.3	3496	160000	720896
2	vltype_struct.h5	669143		5864	663279	0
3	vltype_comp.h5	759872	1.6	3496	35480	720896
4	vltype_struct_comp.h5	468922		5864	463058	0

Table 6: h5stat statistics for random 10000 VL vectors (random data) with the lengths between 1 and 100 stored in HDF5 files with and without compression.

	File name M=10	File size	SR	Metadata size	Raw data size	Unaccountable space
1	vltype.h5	425640	1.9	3496	160000	262144
2	vltype_struct.h5	220830		5864	214966	0
3	vltype_comp.h5	290961	10.6	3496	25321	262144
4	vltype_struct_comp.h5	27394		5864	21530	0

Table 7: h5stat statistics for random 10000 VL vectors (compressible data) with the lengths between 1 and 10 stored in HDF5 files with and without compression.

	File name M=100	File size in bytes	SR	Metadata in bytes	Raw data in bytes	Unaccountable space in bytes
1	vltype.h5	884392	1.3	3496	160000	720896
2	vltype_struct.h5	672130		5864	666266	0
3	vltype_comp.h5	759836	17.7	3496	35444	720896
4	vltype_struct_comp.h5	42848		5864	36984	0

Table 8: h5stat statistics for random 10000 VL vectors (compressible data) with the lengths between 1 and 100 stored in HDF5 files with and without compression.

As one can see from the data in the SR column, structured chunk storage shows savings in all 8 use cases.

We also investigated if storage ratio can be improved by using 4-bytes to store length and offsets values which is a reasonable assumption with the current 4GB limitation for chunk size and upper bound of 4 million for the length of a VL element. Table 9 below shows storage ratios for our experiments. Complete results for 4-byte encoding can be found in Appendix 2. As expected, we can see that 4-bytes values reduce storage size for the cases when no compression is used as shown by the data on lines 1, 5, 9 and 13. Compression provides a comparable storage ratio as shown by the data on lines 3, 7, 11, and 15. Therefore, if desired, 4-byte encoding for offsets and length may be considered as a possible optimization to reduced storage requirements.

		VL element max length	File name	SR 8 bytes	SR 4 bytes
	Random data				
1		10	vltype.h5	1.9	3.0
2			vltype_struct.h5		

		VL element max length	File name	SR 8 bytes	SR 4 bytes
3			vltype_comp.h5	4.2	3.7
4			vltype_struct_comp.h5		
5		100	vltype.h5	1.3	1.5
6			vltype_struct.h5		
7			vltype_comp.h5	1.6	1.6
8			vltype_struct_comp.h5		
	Compressible data				
9		10	vltype.h5	1.9	3.0
10			vltype_struct.h5		
11			vltype_comp.h5	10.6	7.8
12			vltype_struct_comp.h5		
13		100	vltype.h5	1.3	1.5
14			vltype_struct.h5		
15			vltype_comp.h5	17.7	12.8
16			vltype_struct_comp.h5		

Table 9: Storage size ratios for 8 and 4 bytes offset and length values. 4-bytes values help to reduce storage size when no compression is used as shown by the data on lines 1, 5, 9 and 13. Compression provides comparable storage ratio as shown by the data on lines 3, 7, 11, and 15.

The results can be summarized as follows:

1. Using structured chunk for storing VL data shows storage savings in all 8 use cases (see the “File size in bytes” and “SR” columns in the tables above).
2. Since in the current implementation VL data is stored in the global heap, compression is not applied to it as the column “Unaccountable space in bytes”, lines 1 and 3 of each table show.
3. Storing VL data in the structured chunk using compression is beneficial even for randomly generated data as we can see from lines 3 and 4 in Table 5 and Table 6. This is due to the deficiency of the current VL storage described in 2 above.
4. Applying compression to the structured chunk is especially beneficial for compressible raw data as the “File size in bytes” and “CR” columns, lines 3 and 4 in Table 7 and Table 8 show.
5. As Table 9 shows 4-bytes encoding for offsets and lengths provide storage savings when no compression is used on the structured chunk sections. When compression is used storage ratios are comparable with the storage ratios for 8-byte encodings.

4 Conclusions

Our experiments show that proposed structured chunk storage requires comparable or less amount of storage space for both sparse and variable-length data as compared with the current HDF5 capabilities while providing additional benefits of storing data locations for sparse data and enabling compression for VL data.

The new storage will be especially beneficial for storing compressible VL data and compressible sparse data when defined values are in the bounded sub-region (i.e., our main use case for the light sources experimental community).

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”.

References

1. John Mainzer, Elena Pourmal, “RFC: File Format Changes for Enabling Sparse Storage in HDF5”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-File-Format_Sparse_Storage_Changes-2023-07-17.pdf
2. John Mainzer, Elena Pourmal, “RFC: Programming Model to Support Sparse Data in HDF5”, https://github.com/LifeboatLLC/SparseHDF5/blob/main/design_docs/RFC-HDF5-Model-API-Sparse-2023-07-18.pdf
3. 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.

Revision History

December 28, 2023	Version 1 was created for internal review
January 16, 2024	Version 2: updated VL results using 8-byte offset/length encoding

Appendix 1: Benchmarks results for sparse data

This section contains results for `sparse.c` benchmark referenced in section 2. The benchmark was run on macOS Ventura 13.6.2 with the following versions of OS and `clang` compiler:

```
 %uname -v

22.6.0 Darwin Kernel Version 22.6.0: Thu Nov 2 07:43:57 PDT 2023;
root:xnu-8796.141.3.701.17~6/RELEASE_ARM64_T6000

%clang --version

Apple clang version 14.0.3 (clang-1403.0.22.14.1)
Target: arm64-apple-darwin22.6.0
Thread model: posix
```

In the tables below the columns

"Encoded selection size" is the size of the "`selection`" dataset.
"Compressed encoded selection size" is the size of the "`selection_comp`" dataset.

"Sparse storage size" is the size of the "`sparse`" dataset.
"Structured storage size" is the sum of the sizes of the "`selection`" dataset and the "`data`" datasets.

"Compressed sparse storage size" is the size of the "`sparse_comp`" dataset.
"Compressed structured storage size" is the sum of the sizes of the "`selection_comp`" dataset and the "`data_comp`" datasets.

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected locations in each row**
Chunk dimensions:	**1024x1024**
Options of data generation:	**compressible values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	163911	25126	6.5
2	327687	43299	7.6
3	491079	60772	8.1
4	654247	78033	8.4
5	833575	96839	8.6
6	995863	113815	8.7
7	1157623	131018	8.8
8	1318055	148199	8.9
9	1495127	167030	9
10	1654311	184330	9

Table 10: Storage sizes for encoded selection and corresponding storage ratio for randomly selected locations

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	174151	6
2	1048576	348167	3
3	1048576	521799	2
4	1048576	695207	1.5
5	1048576	885799	1.2
6	1048576	1058327	1
7	1048576	1230327	0.9
8	1048576	1400999	0.7
9	1048576	1589335	0.7
10	1048576	1758759	0.6

Table 11: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	32184	25497	1.3
2	58211	43728	1.3
3	82227	61245	1.3
4	100718	78523	1.3
5	118470	97374	1.2
6	123656	114390	1.1
7	134806	131631	1
8	146057	148853	1
9	158131	167728	0.9
10	165582	185066	0.9

Table 12: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected locations in each row**
Chunk dimensions:	**1024x1024**
Options of data generation:	**random values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	163911	25126	6.5
2	327655	43319	7.6
3	491223	60797	8.1
4	654599	77998	8.4
5	833767	96829	8.6
6	995655	113840	8.7
7	1157703	131091	8.8
8	1317671	148267	8.9
9	1495271	166980	9
10	1655047	184264	9

Table 13: Storage sizes for encoded selection and corresponding storage ratio for randomly selected locations

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	174151	6
2	1048576	348135	3
3	1048576	521943	2
4	1048576	695559	1.5
5	1048576	885991	1.2
6	1048576	1058119	1
7	1048576	1230407	0.9
8	1048576	1400615	0.7
9	1048576	1589479	0.7
10	1048576	1759495	0.6

Table 14: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	33287	35377	0.9
2	60040	63815	0.9
3	82625	91533	0.9
4	102831	118979	0.9
5	123207	149079	0.8
6	141821	176330	0.8
7	158999	203826	0.8
8	175973	231247	0.8
9	193669	261224	0.7
10	209636	288753	0.7

Table 15: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected rectangle in the whole chunk**
Chunk dimensions:	**1024x1024**
Options of data generation:	**compressible values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	87	43	2
2	87	43	2
3	87	43	2
4	87	43	2
5	87	42	2.1
6	87	43	2
7	87	43	2
8	87	43	2
9	87	42	2.1
10	87	43	2

Table 16: Storage sizes for encoded selection and corresponding storage ratio for randomly selected rectangle

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	10491	100
2	1048576	20823	50.4
3	1048576	31416	33.4
4	1048576	41703	25.1
5	1048576	52071	20.1
6	1048576	62587	16.8
7	1048576	72987	14.4
8	1048576	83608	12.5
9	1048576	94336	11.1
10	1048576	104416	10

Table 17: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	1657	416	4
2	2405	473	5.1
3	2626	519	5.1
4	1927	537	3.6
5	2959	576	5.1
6	3014	618	4.9
7	2589	658	3.9
8	2613	699	3.7
9	3127	740	4.2
10	2715	780	3.5

Table 18: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected rectangle in the whole chunk**
Chunk dimensions:	**1024x1024**
Options of data generation:	**random values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	87	43	2
2	87	43	2
3	87	43	2
4	87	43	2
5	87	43	2
6	87	43	2
7	87	42	2.1
8	87	43	2
9	87	43	2
10	87	42	2.1

Table 19: Storage sizes for encoded selection and corresponding storage ratio for randomly selected rectangle

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	10491	100
2	1048576	20823	50.4
3	1048576	31416	33.4
4	1048576	41703	25.1
5	1048576	52071	20.1
6	1048576	62587	16.8
7	1048576	72987	14.4
8	1048576	83608	12.5
9	1048576	94336	11.1
10	1048576	104416	10

Table 20: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	12819	10458	1.2
2	23600	20795	1.1
3	34237	31388	1.1
4	45063	41680	1.1
5	54873	52053	1.1
6	65655	62569	1
7	76596	72973	1
8	86900	83600	1
9	98208	94328	1
10	108232	104412	1

Table 21: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected continuous locations in each row**
Chunk dimensions:	**1024x1024**
Options of data generation:	**compressible values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	16455	6693	2.5
2	16439	6726	2.4
3	16423	6687	2.5
4	16455	6706	2.5
5	16423	6710	2.4
6	16439	6707	2.5
7	16407	6730	2.4
8	16455	6762	2.4
9	16423	6697	2.5
10	16455	6720	2.4

Table 22: Storage sizes for encoded selection and corresponding storage ratio for randomly selected continuous locations in each row

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	26695	39.3
2	1048576	36919	28.4
3	1048576	47143	22.2
4	1048576	57415	18.3
5	1048576	68647	15.3
6	1048576	78903	13.3
7	1048576	89111	11.8
8	1048576	99399	10.5
9	1048576	110631	9.5
10	1048576	120903	8.7

Table 23: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	8167	7064	1.2
2	8463	7155	1.2
3	4525	7160	0.6
4	8982	7196	1.2
5	4142	7245	0.6
6	8909	7282	1.2
7	8933	7343	1.2
8	8924	7416	1.2
9	8870	7395	1.2
10	4116	7456	0.6

Table 24: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Maximal percentage of data density:	**10**
Options of data space selection:	**randomly selected continuous locations in each row**
Chunk dimensions:	**1024x1024**
Options of data generation:	**random values**

Percentage	Encoded selection size	Compressed encoded selection size	Storage ratio
1	16455	6693	2.5
2	16455	6687	2.5
3	16455	6723	2.4
4	16439	6702	2.5
5	16439	6728	2.4
6	16439	6698	2.5
7	16439	6725	2.4
8	16439	6768	2.4
9	16423	6696	2.5
10	16407	6713	2.4

Table 25: Storage sizes for encoded selection and corresponding storage ratio for randomly selected continuous locations in each row

Percentage	Sparse storage size	Structured storage size	Storage ratio
1	1048576	26695	39.3
2	1048576	36935	28.4
3	1048576	47175	22.2
4	1048576	57399	18.3
5	1048576	68663	15.3
6	1048576	78903	13.3
7	1048576	89143	11.8
8	1048576	99383	10.6
9	1048576	110631	9.5
10	1048576	120855	8.7

Table 26: Storage sizes for sparse data, emulated structured chunk storage and corresponding storage ratio

Percentage	Compressed sparse storage size	Compressed structured storage size	Storage ratio
1	15073	16944	0.9
2	25909	27183	1
3	36511	37459	1
4	46907	47683	1
5	58399	58978	1
6	68836	69188	1
7	79099	79460	1
8	89446	89748	1
9	100766	100940	1
10	111088	111202	1

Table 27: Storage sizes for compressed sparse data, compressed emulated structured chunk storage and corresponding storage ratio

Appendix 2: VL benchmarks results when using 4 bytes encoding for offset and lengths

This section contains results for `vl_uint.c` benchmark to illustrate the savings when 4-byte integer is used to store offset and length pairs as mentioned section 3. The benchmark was run on macOS Ventura 13.6.2 with the following versions of OS and `clang` compiler:

```
 %uname -v

22.6.0 Darwin Kernel Version 22.6.0: Thu Nov 2 07:43:57 PDT 2023;
root:xnu-8796.141.3.701.17~6/RELEASE_ARM64_T6000

%clang --version

Apple clang version 14.0.3 (clang-1403.0.22.14.1)
Target: arm64-apple-darwin22.6.0
Thread model: posix
```

	File name M=10	**File size**	**SR**	**Metadata size**	**Raw data size**	**Unaccountable space**
1	vltype.h5	425640	3.0	3496	160000	262144
2	vltype_struct.h5	140847		5864	134983	0
3	vltype_comp.h5	290982	3.7	3496	25342	262144
4	vltype_struct_comp.h5	79020		5864	73156	0

Table 28: h5stat statistics for random 10000 VL vectors (random data) with the lengths between 1 and 10 stored in HDF5 files with and without compression; offset and length are stored as 4 bytes integers.

	File name M=100	**File size**	**SR**	**Metadata size**	**Raw data size**	**Unaccountable space**
1	vltype.h5	884392	1.5	3496	160000	720896
2	vltype_struct.h5	589143		5864	583279	0
3	vltype_comp.h5	759872	1.6	3496	35480	720896
4	vltype_struct_comp.h5	485597		5864	479733	0

Table 29: h5stat statistics for random 10000 VL vectors (random data) with the lengths between 1 and 100 stored in HDF5 files with and without compression; offset and length are stored as 4 bytes integers.

	File name M=10	**File size**	**SR**	**Metadata size**	**Raw data size**	**Unaccountable space**
1	vltype.h5	425640	3.0	3496	160000	262144
2	vltype_struct.h5	140830		5864	134966	0
3	vltype_comp.h5	290961	7.8	3496	25321	262144
4	vltype_struct_comp.h5	37106		5864	31242	0

Table 30: h5stat statistics for random 10000 VL vectors (compressible data) with the lengths between 1 and 10 stored in HDF5 files with and without compression; offset and length are stored as 4 bytes integers.

	File name M=100	**File size in bytes**	**SR**	**Metadata in bytes**	**Raw data in bytes**	**Unaccountable space in bytes**
1	vltype.h5	884392	1.5	3496	160000	720896
2	vltype_struct.h5	592130		5864	586266	0
3	vltype_comp.h5	759836	12.8	3496	35444	720896
4	vltype_struct_comp.h5	59550		5864	53686	0

Table 31: h5stat statistics for random 10000 VL vectors (compressible data) with the lengths between 1 and 100 stored in HDF5 files with and without compression; offset and length are stored as 4 bytes integers.