RFC: New Requirements for HDF5 Chunk Cache

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

Introduction of the structured chunk storage to support sparse data in HDF5 [1, 2] requires rework of some components of the HDF5 library, specifically, rework of chunk cache: chunk cache has to support I/O on a chunk that may have one or more sections with different kinds of data stored in each section. Since we will be rewriting this component, it is our opportunity to address not only I/O for regular and structured chunks but also to address known deficiencies of the current chunk cache implementation.

The purpose of this document is to summarize new requirements for HDF5 chunk cache and to solicit input from the HDF5 developers' community before finalizing chunk cache design.

New chunk cache implementation to support sparse storage and to address known deficiencies will be contributed to the open source HDF5 software maintained by The HDF Group.

1 Introduction

Lifeboat, LLC has been working on designing sparse data storage in HDF5.

Sparse data is common in many scientific disciplines and experiments. Several examples are discussed in [3], including High Energy Physics and Neutron and X-ray Scattering. In these examples, only 0.1% to 10% of gathered data is of interest. HDF5, due to its proven track record and flexibility, remains the data format of choice. As the amount of data produced continues to grow due to higher instrument resolution and higher sampling rates, there is a demand for efficient management of sparse data in HDF5.

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), may result in a huge memory footprint. Therefore, a different approach for sparse data management is needed. The proposed approach was first prototyped in [3], and outlined next to provide some context to the requirements discussed in section 2.

As it is impractical to hold the entire sparse dataset in memory, we break the sparse dataset into user-specified, regular, n-dimensional hyper-rectangles. A sparse chunk is a hyper-rectangle endowed with an HDF5 selection, which represents all defined (i.e., user-supplied) entries in its domain. This way, each sparse chunk contains a selection (data coordinates) and associated data. This approach allows us to store only data of interest and to operate on sparse chunks using existing HDF5 facilities for serialization and deserialization, and for constructing partial and parallel I/O on sparse data.

We generalized the idea of sparse chunk described above and designed a new storage layout called Structured Chunk. Like a regular chunk, a structured chunk will be used to store the values contained in some n-dimensional rectangular volume in a dataset. However, unlike regular chunks, it will be composed of two or more sections, which in combination, will describe the values contained in the volume. For sparse chunk that contains data of fixed-size datatype (e.g., HDF5 numeric type) sparse chunk will have two sections: one section to store encoded coordinates of the defined values and another section to store the defined values themselves. When storing sparse data of variable-length datatype sparse chunk will have three sections: a section to store coordinates, a section with the defined values, and a section with the pointers to the defined values. As with the regular chunked layout the structured chunk layout will support partial and parallel I/O, and data filtering that can be applied to different sections of the structured chunk.

We have already outlined the extensions to the HDF5 file format [1] and public APIs [2] to structured chunk storage. Now we are in the process of designing the HDF5 library changes keeping in mind that in the future, structured chunk can be used not only for storing sparse data of any datatype including

variable-length datatype (e.g., strings) but also for storing dense variable-length data and non-homogenous data arrays (e.g., arrays in which each element has its own datatype).

Since chunk cache is one of the most important components of the library and has to work with both dense and structured chunks, we started our design for the HDF5 library changes with designing new chunk cache.

The following sections discuss motivations for new design and summarize the requirements. We would like to make sure that we have a complete list of requirements before proceeding to the design task. We are looking for input from the HDF5 developers' community. *New chunk cache implementation will be contributed to the open source HDF5 software maintained by The HDF Group.*

2 Chunk Cache Requirements

This section summarizes motivations for chunk cache re-implementation and outlines[1] requirements for new implementation.

There are two major drivers for re-implementing chunk cache: support for new type of chunk – structured chunk, and known deficiencies of the current cache design and implementation. Next subsections outline requirements for each mentioned driver.

2.1 Support for Structured Chunk

The addition of support for sparse data (and therefore structured chunks) will require significant awareness of structured chunks on the part of the chunk cache – in particular, it must:

1. *Cache un-encoded versions of sections of structured chunks.* The initial application of this ability will be the selections associated with structured chunks used to store sparse datasets. In the future we will need to add un-encoded versions of pointers into variable-length data heaps for sparse and dense variable-length datasets and tables of types used in non-homogeneous data sets.
2. *Be able to adapt to changing structured chunk section sizes*. For example, in the sparse chunk case, both the encoded selection and the fixed data sections will change size as entries in the structured chunks are defined and undefined. Similarly, for variable length data, heap size will change as variable length data is added, deleted, or modified.
3. *Be able to perform operations on individual sections of a structured chunk both on load and in preparation for write.* For example, for sparse data, the selection section will have to be decoded on read, and encoded prior to write. Similarly, it will be useful to defrag heap sections just prior to filtering and writing to disk if the percentage of unused space exceeds some user defined value.
4. *Run different filter pipelines on the different sections of structured chunks on read.* Similarly, on write, run different filter pipelines on the various sections of the structured chunk, and assemble the resulting buffers into the *on-disk* image of the structured chunk on write.

[1] This document is work-in-progress and listed requirements are preliminary.

5. *Manage checksums on individual sections of structured chunks where required.* This is necessary, as sections of structured chunks that indicate where raw data is to be found (i.e., the encoded selections used in sparse data) are in effect metadata, whose correctness must be verified to prevent buffer overflows and related run time errors.
6. *Be able to read and cache individual sections of structured chunks.* A possible application is to cache only the selection sections of structured chunks in sparse datasets, so as to allow efficient determination of the locations of defined data across the entire dataset. Whether this will prove a useful optimization remains to be seen, but if we want to explore it, the capability should be designed into the chunk cache to begin with. The facility could also be used to support reads / writes of individual pieces of data without loading the full structured chunk in the un-filtered case – as is currently done with contiguous and unfiltered chunked data sets.

2.2 Addressing current deficiencies

The existing implementation has a number of problematic design decisions. One of them is the decision to create one chunk cache per open chunked data set. When large numbers of datasets must be held open, this results in a large allocation of RAM to the chunk caches, the vast majority of which is un-used at any point in time. Very often this leads to a poor performance due to memory swapping.

Further, by default, the library creates a 1 MiB chunk cache for each opened chunked data set. This results in thrashing if the dataset in question was created with a chunk size greater than 1 MiB and data filtering enabled, and if the I/O request is not aligned with logical chunk boundaries.

Finally, the indexing method used by the current chunk cache uses a hash table for indexing. While this is common enough, the hash table handles collisions by evicting the pre-existing entry. To minimize the chances of pathological situations resulting from this design decision, the hash table is made larger than would otherwise be necessary. While this seems to control the problem, it would be good to resolve it completely.

The following requirements should be fulfilled to address current chunk cache deficiencies regardless of the chunk type.

1. *Shared chunk cache:* To avoid the chunk cache size explosion when large numbers of datasets are open simultaneously, share the chunk cache between all open datasets. Whether we do this on a per file basis, or across all open datasets in all open files is an open question. Given the complexities of a global chunk cache, we currently lean towards a per file chunk cache – but the idea should be explored.
2. *Track which chunks belong to which datasets.* We need this to facilitate flushing individual datasets, both on user command and on close. It also facilitates replacement policies based on datasets as well as individual chunks.
3. *Support for minimum chunk cache space allocations on a per dataset basis.*
4. *Support for alternate replacement policies.* While the initial replacement policy will almost certainly be some variation on LRU, it is easy to come up with scenarios where alternate policies would be useful. Support for this should be designed in.

5. *Support for adaptive chunk cache resizing.* The metadata cache has had the ability to adjust its size to match the current working set size within user specified limits and time frames since HDF5 version 1.8. Given the success of this feature, and how the working set size of the chunk cache can vary based on access patterns, adding a similar facility to the chunk cache seems obvious.
6. *Support for background thread(s) prefetching chunks, or preparing dirty chunks for flush when they become likely candidates for flush and/or eviction.* Given the cost of filtering large chunks, this should provide a significant performance boost when there is available compute to support it.
7. *Multithread support.* While the version of the chunk cache for the existing HDF5 library will have to be single thread, the Lifeboat version will have to be multi-thread to fit into the H5+ product.
8. *Make chunk cache available for parallel I/O.*
   To do this, move some or all support for collective I/O on chunked datasets with variable-size chunks to the chunk cache and/or the chunk format layer.

   This capability already exists in HDF5 to allow (collective only) writes to filtered chunked datasets in the parallel build. As the algorithm is essentially the same for any dataset with variable size chunks, generalizing it and moving it to chunk cache and/or the chunk format layer would allow this code to be shared between all such types of chunked datasets – present and future.

3 Final Recommendation for Chunk Cache Implementation in HDF5

To be added after discussions with the HDF5 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 12, 2023:	Version 1 was created for internal reviews.