Variable Length Data in HDF5 Sketch Design

Quincey Koziol
John Mainzer

7/15/19

INTRODUCTION

As demonstrated in the benchmarks described in the appendix, the existing implementation of variable length data in HDF5 has significant performance problems.

In this paper, we outline the current method of storing variable length data, discuss the reasons for its poor performance, and offer a sketch of a proposed re-implementation.

Scot Breitenfeld wrote the above mentioned benchmark to comparing performance of the current variable length data implementation with a mockup of the proposed re-implementation. While the benchmark does not exactly mimic the structure of the proposed re-implementation, the cases are sufficiently similar as to suggest significant performance gains

CURRENT IMPLEMENTATION OF VARIABLE LENGTH DATA IN HDF5

The current implementation of variable length data types stores each variable length datum in the global heap, and a reference to the datum in the dataset proper.

This architecture has several problematic features:

1) The global heap is treated as metadata. Thus variable length data reads and writes put pressure on the metadata cache, and potentially force the eviction of metadata. Thus in the typical EOD use case where large amounts of variable length raw data are written in a short period of time, metadata proper may be squeezed out of the metadata cache, with the resulting performance degradation.

2) Variable length data is not segregated by source dataset in the global heap. Thus typical EOD use cases in which writes to multiple datasets are interleaved, will result in related data being scattered through the global heap, resulting in obvious read inefficiencies.

3) The global heap does not support the filter mechanism, making it impossible to compress variable length data.

4) The global heap is designed to support the re-use of freed global heap space. As a general rule, EOD data does not require this capability, but it must still pay for it – primarily through space and time inefficiencies.

5) Due to the requirement that all operations that modify metadata be collective and generate the same stream of dirty metadata on each process, practically speaking, variable length data may only be written in the serial version of HDF5.

Scot Breitenfeld performed a number of benchmarks (see appendix), comparing I/O to vectors of variable length data with reads and writes of the same variable length data flattened and written to a fixed size vector. To facilitate reads in the latter case, the lengths (but not the offsets) of the flattened variable length data were stored in a second vector.

These benchmarks show the performance costs of using the existing variable length data implementation, and the potential performance gains of flattening variable length data into a vector. The following proposed re-implementation builds on this observation.

PROPOSED RE-IMPLEMENTATION OF VARIABLE LENGTH DATA IN HDF5

At a conceptual level, the proposed re-implementation of variable length data in HDF5 associates a unique, secondary, extensible vector of bytes with every data set that contains variable length data. Each variable length datum is stored in the extensible vector of bytes, and represented in the primary data set by the offset and length of the variable length datum in the extensible vector.

Space in the extensible vector is allocated by extending the vector. Thus in EOD use case in which data is stored as it is generated, temporally adjacent variable length data is stored in adjacent locations in the vector – allowing efficient sequential data writes and reads.

In the EOD case, data is seldom if ever modified after it is written. Thus in this use case, there is no need for any provision for re-using space in the extensible vector when variable length data is added, deleted, or re-sized. However, this capability is desirable in other use cases, and is easily supported by optionally associating a free space manager with the extensible vector. Since this augmentation optional, it imposes no additional overhead on EOD use cases.

In his benchmarks (appended to this document), Scot Breitenfeld compared I/O performance to a vector of variable length data with that of the same data flattened into a vector with the lengths (but not the offsets) stored in a second vector. While his benchmark does not exactly mimic the proposed reimplementation, the differences are minor, and the speedups are significant. Further, his benchmark only writes one dataset at a time, and thus avoids the interleaving effect discussed in the previous section. If there is sufficient interest in variable length data performance in HDF5, his results are sufficient to warrant a full design effort and subsequent implementation.

IMPLEMENTATION DETAILS TO BE ADDRESSED

While the basic concept of the proposed re-implementation of variable length data in HDF5 should be clear enough at this point, the devil is always in the details. As this document is only a sketch design, we content ourselves with pointing out major issues to be addressed, and suggesting likely solutions. Detailed investigation of these design issues is beyond the scope of this paper, and must be addressed in a subsequent RFC.

TRANSPARENCY TO THE USER

On the one hand the extensible vector of bytes should be hidden from the user as an extra data structure that is created when the primary data set is created.

On the other hand, we must allow the user to configure the extensible vector of bytes to maximize performance given the particulars of the data set. Thus, for example, typical length and composition of variable length data will have an effect on the optimal chunk size and filter pipeline.

At a guess, we will square this circle by providing generally workable defaults, and the ability to adjust them via the DCPL.

REPRESENTATION OF THE EXTENSIBLE ARRAY OF BYTES

Absent a major re-design of the HDF5 library, the only generally plausible way of implementing the extensible vector of bytes is as a chunked 1 D data set. Management of the chunks requires an index – most likely the existing extensible array.

This decision implies that we must provide a mechanism for selecting the chunk size of the extensible vector of bytes, and also for configuring the associated filter pipeline if desired.

The decision to use a chunked representation also raises the question of chunk caching. At present, we create a separate chunk cache for each open chunked data set. While this has proved problematic, particularly with applications that open large numbers of datasets, it is the obvious default in this case. Note, however, that it implies the creation of yet another switch to allow the user to select the size of the chunk cache for the extensible vector of bytes.

On the other hand, re-working the chunk cache is one of the issues being considered in the sparse chunks RFC – and thus best we postpone this issue for now.

VARIABLE LENGTH DATA IN PARALLEL

In parallel HDF5, we require that all operations that modify metadata be collective, so that all metadata caches see the same sequence of dirty metadata. Since the current implementation of variable length data stores it as metadata, this requirement makes is effectively impossible to write variable length data in parallel HDF5.

While we will probably not implement it immediately, the proposed re-implementation of variable length data offers the possibility of writing variable length data in parallel, albeit only as collective write. We could do this as follows:

1) On each process, compute the total length of the variable length data to be written.
2) Scatter / gather the values from 1) above, so that each process knows the amount of variable length data every process needs to write.
3) On each process, compute the total number of bytes of variable length data to be written, and collectively extend the extensible vector of bytes accordingly.
4) On each process, compute the number of bytes of variable length data to be written by lower rank processes. Write that processes variable length data to the extensible vector of bytes starting at the initial length of the vector plus this sum. Update the primary dataset to reflect the assigned locations of the variable length data.
5) Write the primary data set.

Needless to say, this algorithm could be extended to allow compression via techniques similar to those currently used to implement compression in parallel.

Note also that this algorithm makes no allowance for free space management in the extensible vector of bytes. As all processes would have to participate in space allocation/deallocation for each variable length datum, the overhead would likely make this impractical. As mentioned above, this point is likely moot for EOD use cases – but possibly a significant limitation elsewhere.

Finally, observe that this approach is not applicable to compact datasets, as they store raw data in the object header, and thus suffer the same problems as the existing variable length data implementation in the parallel case.

INTERACTION WITH COMPACT DATASET LAYOUT

Implementing the extensible vector of bytes as a chunked dataset would seem to largely defeat the purpose of compact datasets – although certainly no worse than the existing implementation. If this becomes an issue, alternate implementations such as storing each variable length datum as a separate object header message may be worth considering.

ACKNOWLEDGEMENTS

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

SPARSE STORGE USING HDF5

SPARSE STORAGE USE CASES

The issue of sparse data is an important issue within the fields of science and engineering, and the storage format of the sparse data often plays a critical role in I/O performance. One option in storing sparse data is to flatten the non-zero entries into a 1D variable length (VL) dataset, where each variable length entry would be contiguous sections of nonzero element values of the sparse matrix. The following sections in this paper presents the penalty in using VL datasets over a more typical array dataset representation in an HDF5 file.

BENCHMARKING VARIABLE LENGTH I/O

THIS STUDY REPORTS HDF5'S I/O PERFORMANCE DIFFERENCE BETWEEN USING A 1D DATASET AND A 1D DATASET COMPOSED OF VARIABLE LENGTH (VL) DATATYPES. THE VL DATA STORED IN AN ARRAY OF NATIVE DATATYPE, ABSTRACTLY REPRESENTED AS A STANDARD 2D ARRAY (FIGURE 1A), WHERE EACH ROW OF THE ARRAY IS A VL ENTRY, WAS STORED IN A 1D DATASET OF SIZE **M**X**N**. FOR VL DATATYPE STORAGE, THE VL DATASET'S LENGTH IS CONSIDERED TO BE: (1) THE SAME FOR EACH ROW,

FIGURE 1(B), OR A FUNCTION OF THE ROW NUMBER,

Figure 1(c). The I/O size was increased by increasing ***m*** while ***n*** is held constant at 4096 for cases (a) and (b). For case (c), the number of rows was doubled, and the VL increases and decreases linearly as a function of the row number; resulting in the same amount of I/O as the (a) and (b) cases.

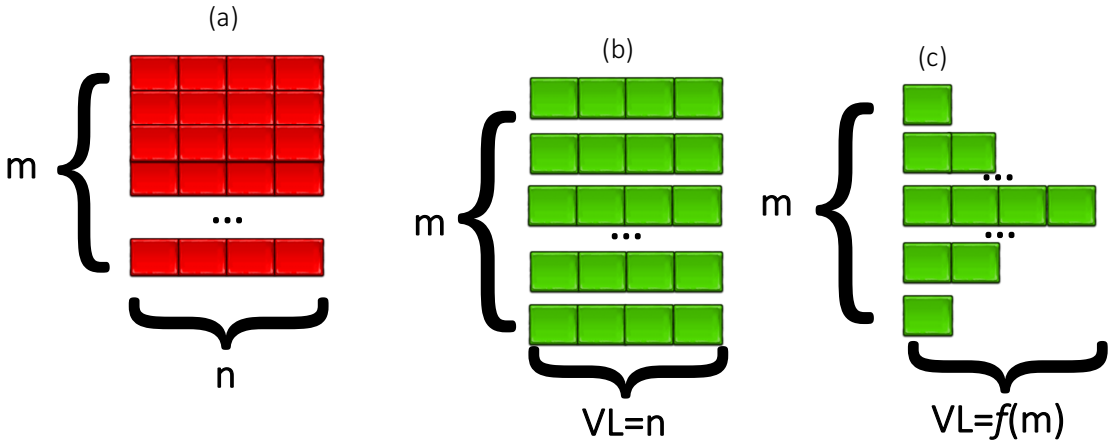

FIGURE 1 (A) NATIVE DATATYPE DATASET, (B) CONSTANT VL DATASET, AND (C) VL DATASET.

Benchmarking programs run in four stages, where each stage consists of a single and separate run. The four steps are: (1) Create and write a 1D dataset, (2) create and write a VL dataset, (3) open the HDF5 file and read the 1D dataset and (4) open the HDF5 file read the 1D VL dataset. Unless noted otherwise, these four stages were run eight times and the average times are reported in the subsequent sections.

The VLtest/***t_vlen_bench.c*** program can be found at https://github.com/brtnfld/mpi_io.git and implements the storing of VL datatypes, Figures 1b and 1c. The VLtest/**t_pseudo_vlen_bench.c** uses a dataset of the native datatype to store the VL data, Figure 1a. Additionally for this case, Figure 1a, the length of each variable is stored in a sperate 1D dataset of size ***m***, and the timing results presented below also include the IO associated with this 1D dataset.

CASE 1

The first case tries to minimize the effect of the Linux file system cache. The HDF5 library was modified by adding the option ***O_DSYNC*** to the POSIX ***open*** function in *H5FDsec2.c*. Additionally, the utility ***nocache***[1] was used to disable cache generation by the benchmark program. The I/O was to a non-RAID partition (ext4) on a standard hard drive (WDC WD5003ABYX-18WERA0) and for a Linux 3.10 kernel. The final times reported were the average over ten runs, Figure 2.

Dataset Size (MiB)	Write (s)	Read (s)	VL Write (s)	VL Read (s)
64	0.45	0.24	43.0	0.43
128	0.65	0.36	83.6	0.66
256	1.12	0.81	168.0	1.23
512	2.06	1.82	335.1	2.19
1024	3.78	3.48	670.7	4.05
2048	7.23	7.74	1343.9	7.59
4096	14.4	14.6	2712.7	15.5

[1] https://github.com/Feh/nocache

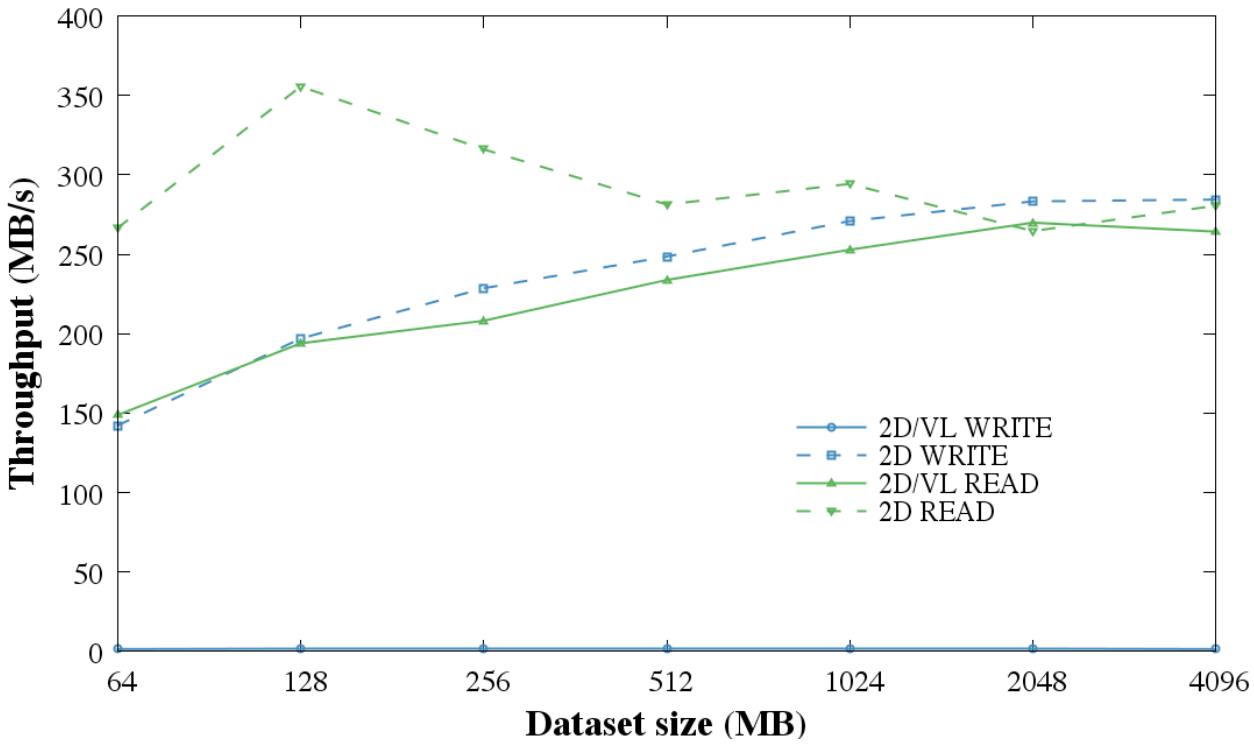

FIGURE 2 VL I/O THROUGHPUT TO A STANDARD HARDRIVE FOR A NON-PARALLEL FILESYSTEM WITH NO CACHING EFFECTS.

CASE 2

The remaining cases in this report study the performance I/O is to a Lustre file system. In contrast to Case 1, the HDF5 library was not modified and the utility ***nocache*** was not used. The stripe size was 16MB, and the stripe count was 12 on Edison (NERSC).

Dataset Size (MiB)	Write (s)	Read (s)	VL Write (s)	VL Read (s)
64	0.072	0.036	0.17	0.13
128	0.12	0.078	0.25	0.25
256	0.26	0.16	0.51	0.51
512	0.51	0.31	1.00	1.01
1024	1.09	0.60	2.07	2.03
2048	2.22	1.21	4.22	4.16
4096	7.63	2.39	9.54	8.60

CASE 3

The third case used the same parameters as case 2, except page buffering was enabled for two different page sizes. Cases 2 and 3 are summarized in Figure 3 and Figure 4.

1kiB page size.

Dataset Size (MiB)	Write (s)	Read (s)	VL Write (s)	VL Read (s)
64	0.070	0.038	0.26	0.16
128	0.12	0.077	0.50	0.34
256	0.24	0.15	0.94	0.67
512	0.50	0.29	1.86	1.34
1024	1.00	0.59	3.72	2.70
2048	2.07	1.15	7.32	5.41
4096	3.97	2.38	15.16	11.36

1 MiB page size.

Dataset Size (MiB)	Write (s)	Read (s)	VL Write (s)	VL Read (s)
64	0.074	0.039	0.13	0.13
128	0.12	0.079	0.30	0.26
256	0.25	0.16	0.53	0.52
512	0.62	0.30	1.05	1.02
1024	1.05	0.61	3.3	2.08
2048	2.12	1.20	6.36	4.18
4096	4.24	2.38	14.0	8.70

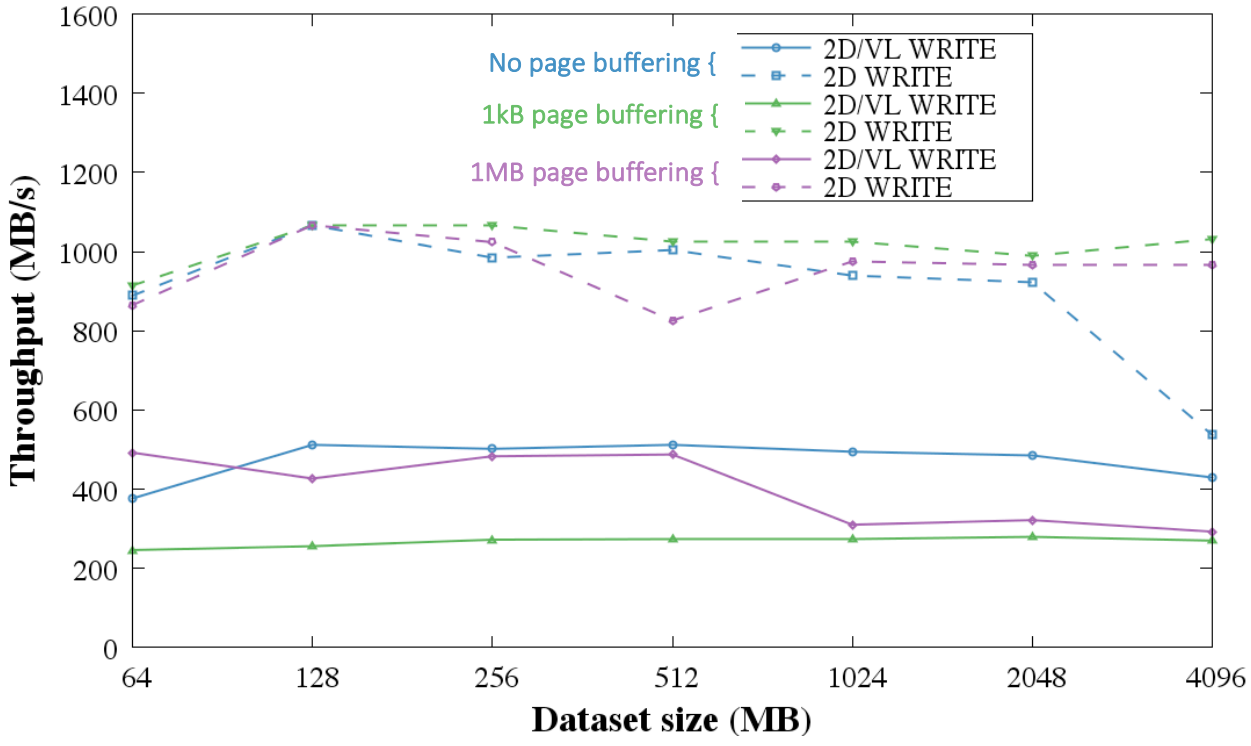

FIGURE 3 WRITE BANDWIDTH ON LUSTRE.

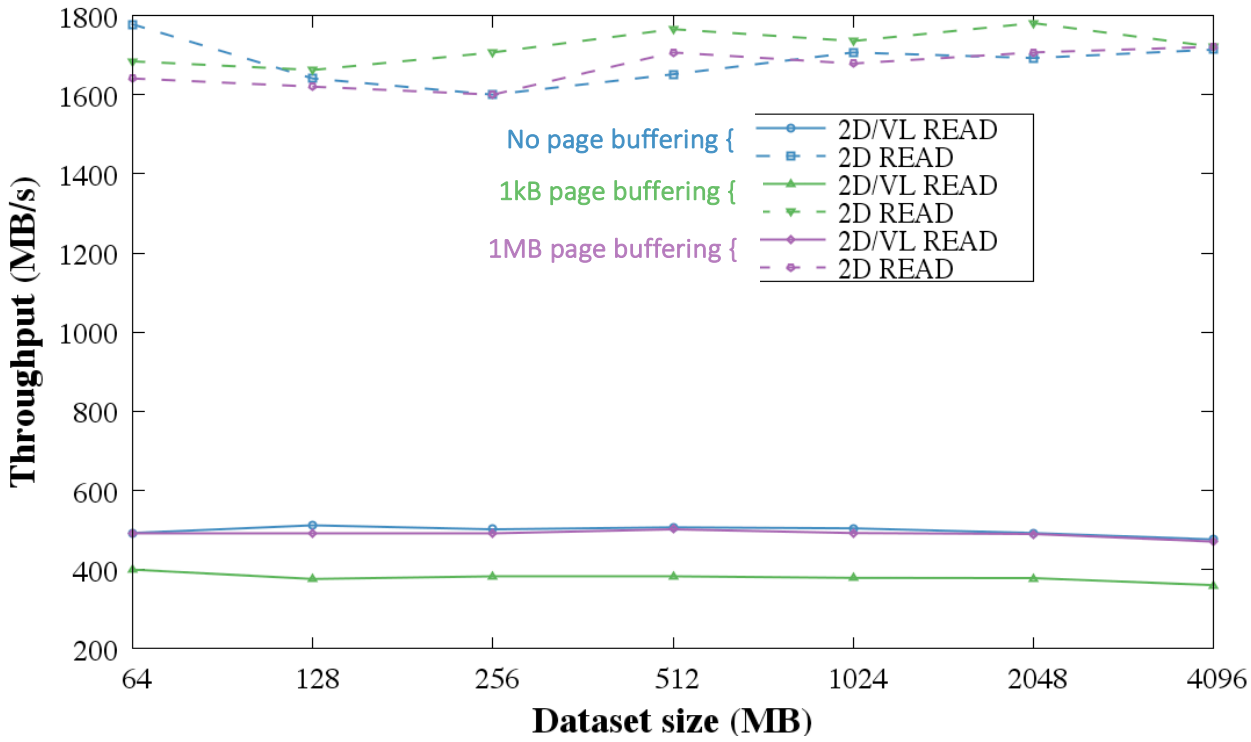

FIGURE 4 READ BANDWIDTH ON LUSTRE.

CASE 4

THIS CASE USED VARIABLE LENGTH DATA FOR WHICH THE LENGTH INCREASES AND THEN DECREASES LINEARLY AS A FUNCTION OF THE ROW,

Figure 1c. The amount of data I/O is the same as in case 3. Using a constant length VL data has marginally better I/O performance when compared to using varying length VL data.

Dataset Size (MiB)	Write (s)	Read (s)	VL Write (s)	VL Read (s)
64	0.072	0.040	0.31	0.25
128	0.12	0.077	0.70	0.50
256	0.25	0.15	1.30	1.00
512	0.50	0.30	2.55	2.02
1024	1.01	0.61	5.10	4.16
2048	2.00	1.16	10.48	8.36
4096	4.01	2.35	20.61	16.95

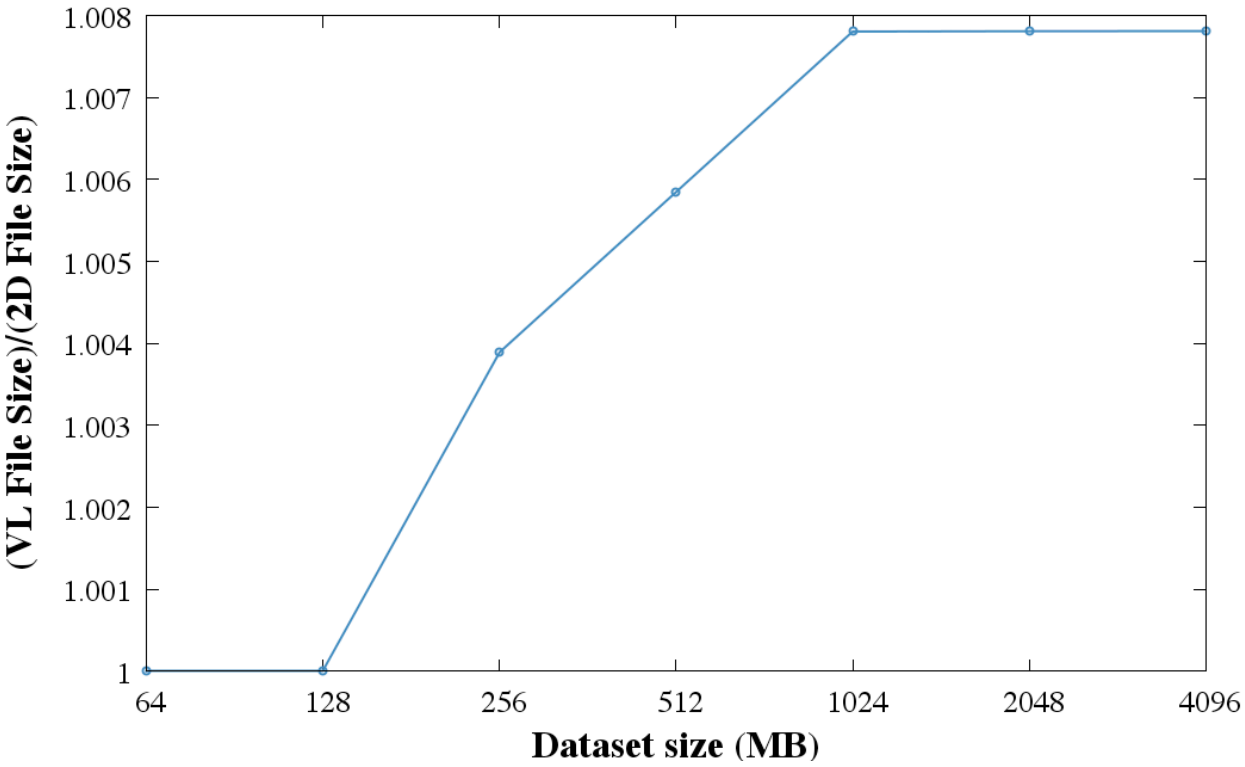

FIGURE 5 CASE 4 FILE SIZE RATIO COMPARISON WHEN STORING A VL DATASET AND STORING A NATIVE DATATYPE DATASET.

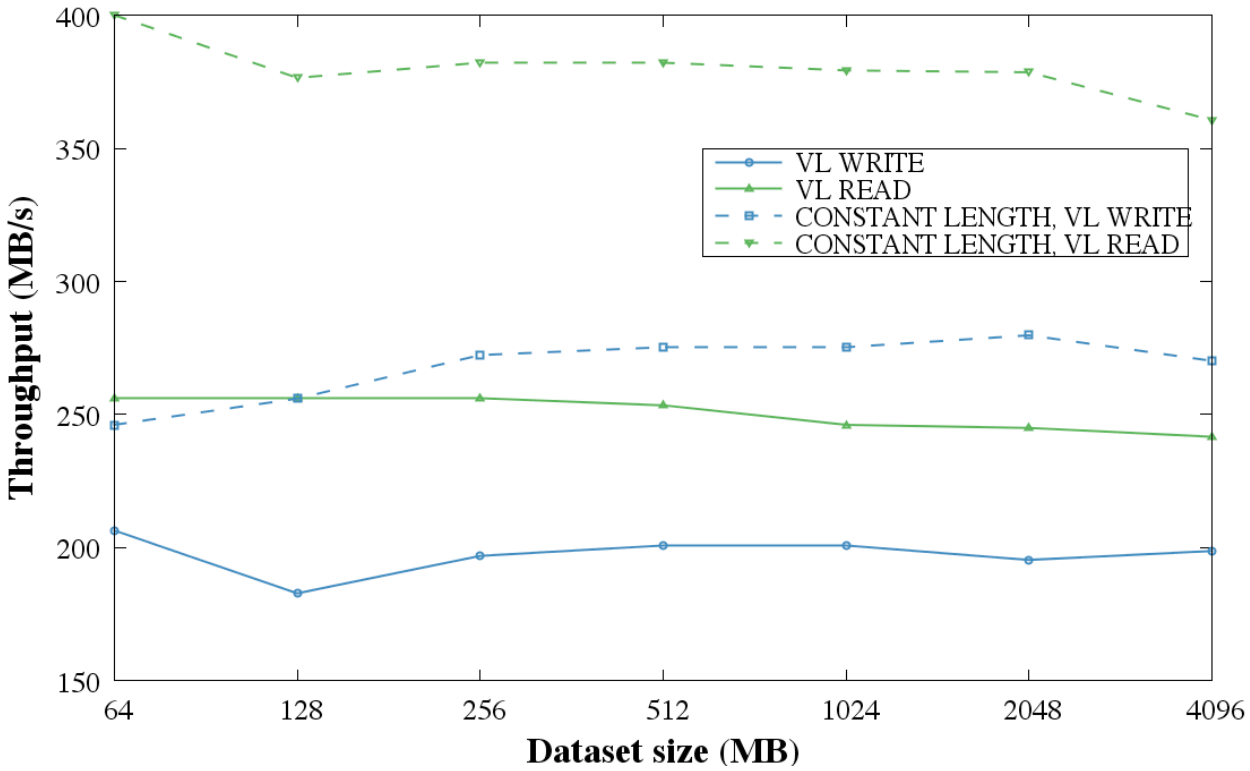

FIGURE 6 I/O COMPARISION BETWEEN CONSTANT (CASE B) AND VARIABLE LENGTH DATA (CASE C).

CASE 5

THIS CASE USED VARIABLE LENGTH DATA DETAILED IN CASE 4,

Figure 1c, to study the effects of page buffering. The amount of data I/O is the same as in case 4. The file space page size was set to 1 MiB, and the page buffer size was set to multiples of the file space page size. The minimum metadata and raw data percentage was set to zero. Figure 7 to Figure 10 report the effects of page buffer for (1) writing native datatype dataset, (2) reading a native datatype dataset, (3) writing VL dataset and (4) reading a VL dataset.

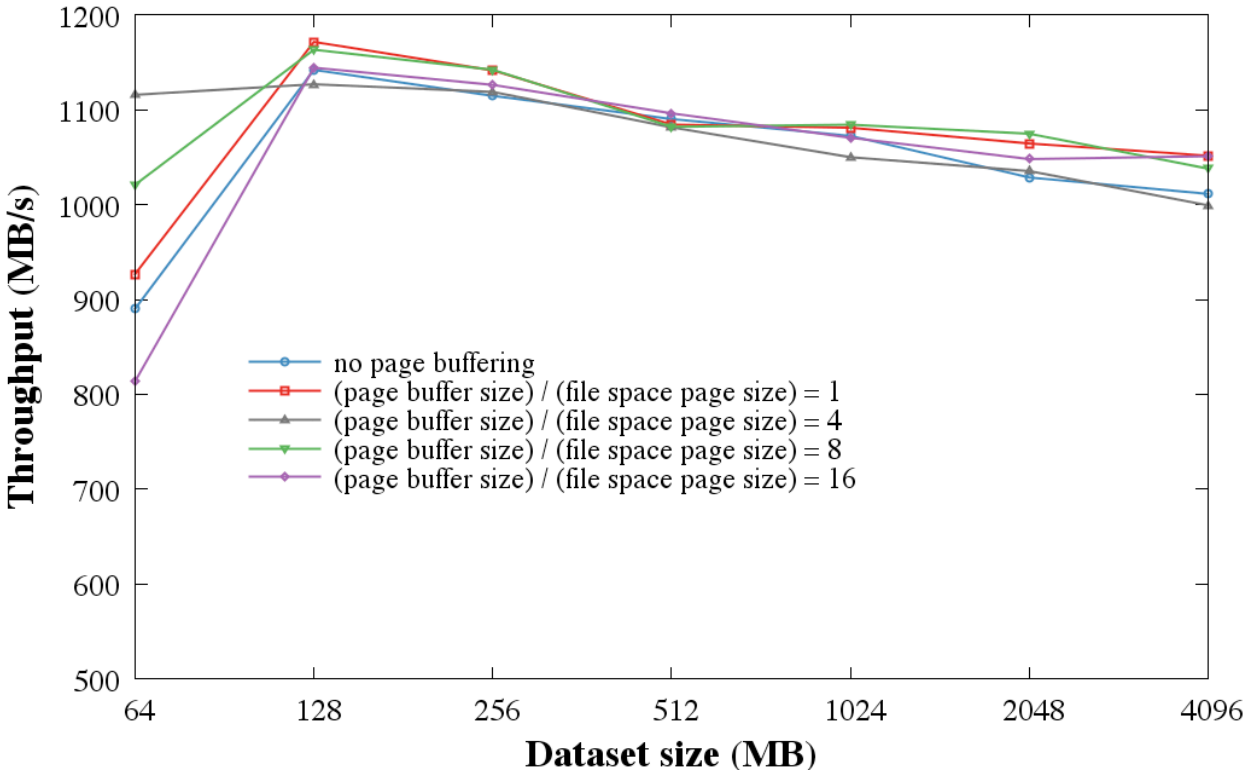

FIGURE 7 EFFECTS OF PAGE SIZE FOR NATIVE DATATYPE DATASET **WRITE**.

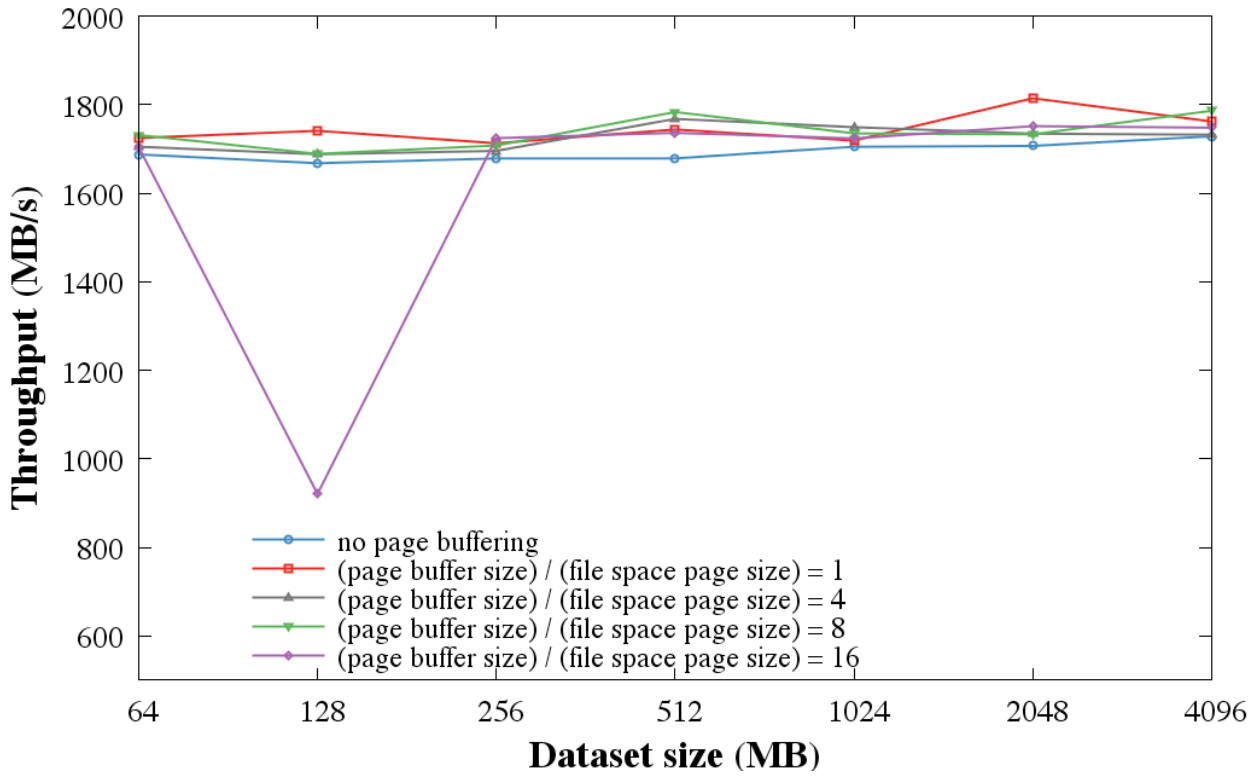

FIGURE 8 EFFECTS OF PAGE SIZE FOR NATIVE DATATYPE DATASET **READ**.

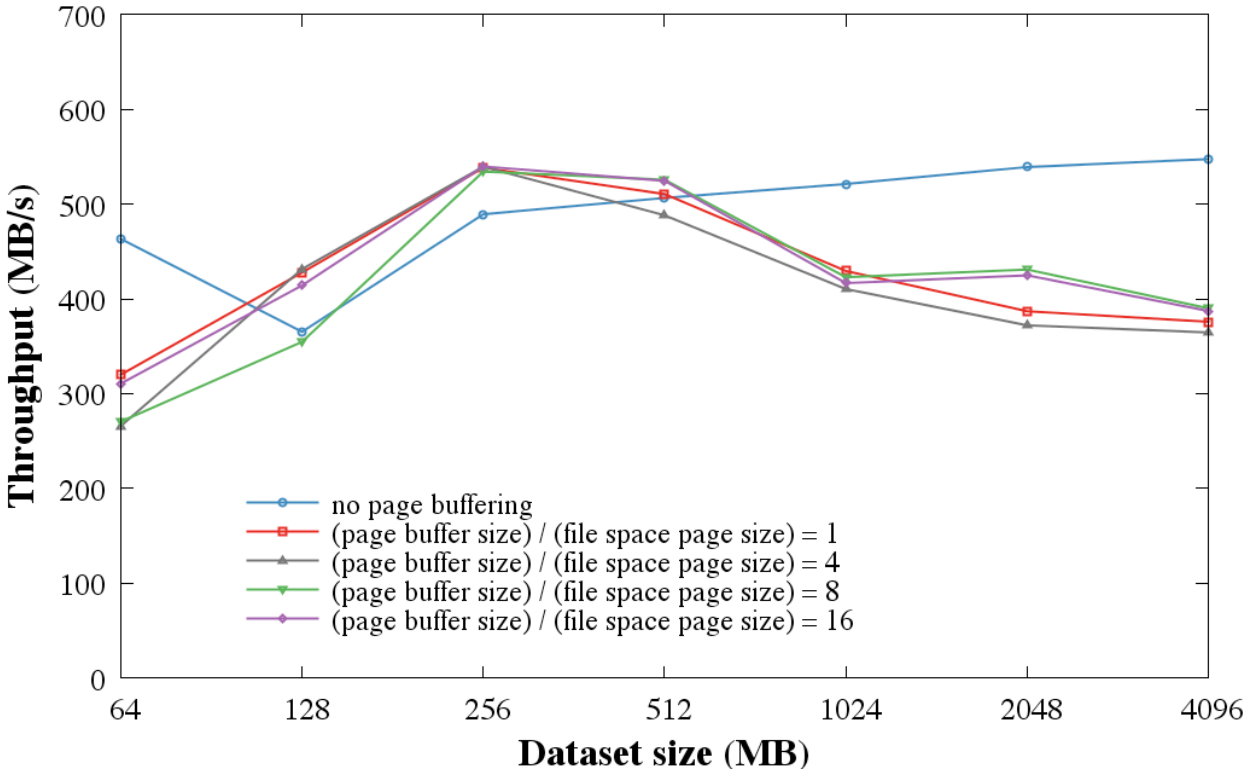

FIGURE 9 EFFECTS OF PAGE SIZE FOR VL DATASET **WRITE**.

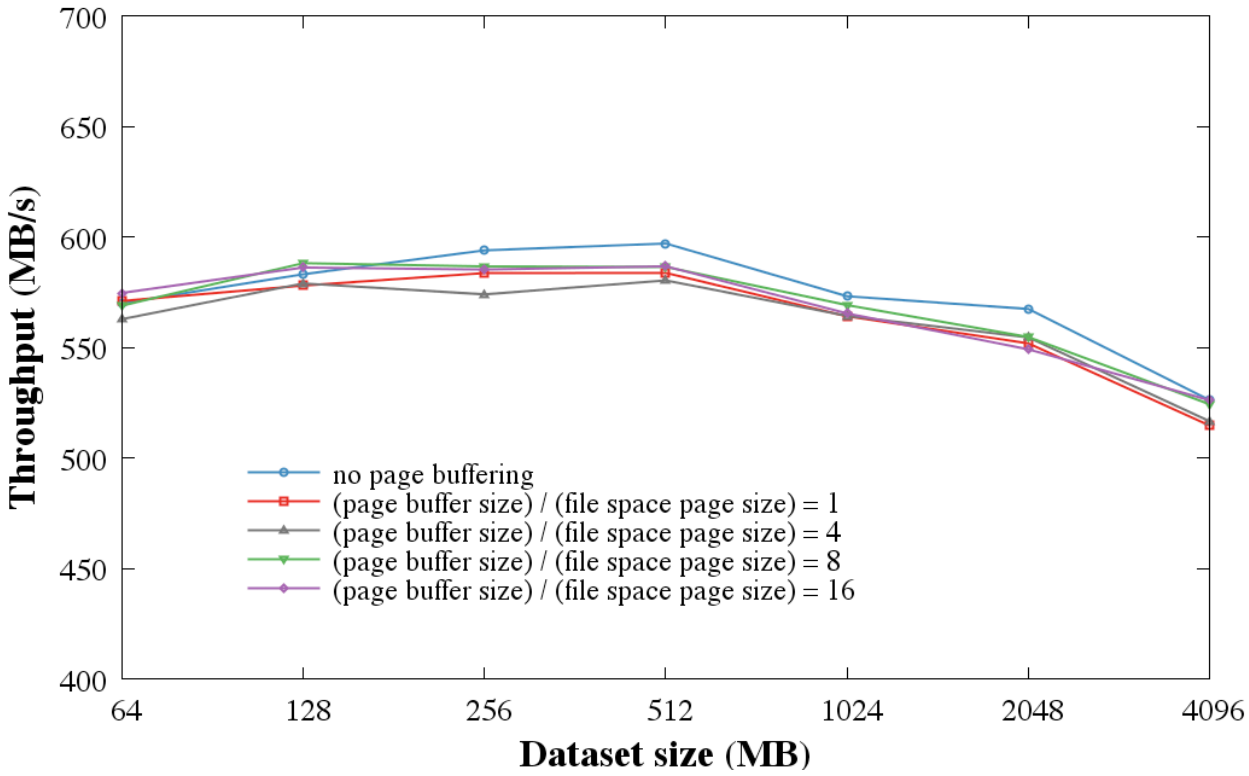

FIGURE 10 EFFECTS OF PAGE SIZE FOR VL DATASET **READ**.