A Storage Framework for Managing Scientific Data[*]

Souvik Bhattacherjee
Department of Computer Science, University of Maryland, College Park
bsouvik@cs.umd.edu

ABSTRACT

In this paper, we present the design, implementation, and evaluation of *PStore*, a no-overwrite storage framework for managing large volumes of array data generated by scientific simulations. PStore consists of two modules, a data ingestion module and a query processing module, that respectively address two of the key challenges in scientific simulation data management. The data ingestion module is geared toward handling the high volumes of simulation data generated at a very rapid rate, which often makes it impossible to offload the data onto storage devices; the module is responsible for selecting an appropriate *compression scheme* for the data at hand, chunking the data, and then compressing it before sending it to the storage nodes. On the other hand, the query processing module is in charge of efficiently executing different types of queries over the stored data; in this paper, we specifically focus on *dicing* (also called *range*) queries. PStore provides a suite of compression schemes that leverage, and in some cases extend, existing techniques to provide support for diverse scientific simulation data. To efficiently execute queries over such compressed data, PStore adopts and extends a two-level chunking scheme by incorporating the effect of compression, and hides expensive disk latencies for long running range queries by exploiting chunk prefetching. In addition, we also parallelize the query processing module to further speed up execution. We evaluate PStore on a 140 GB dataset obtained from real-world simulations using the regional climate model CWRF [5]. In this paper, we use both 3D and 4D datasets and demonstrate high performance through extensive experiments.

1. INTRODUCTION

High-resolution physical simulations are growing in importance in a variety of scientific domains, and are often the only tool capable of providing useful predictions. For instance, in mesoscale climate modeling, high-resolution simulations can be used to predict the effects of various processes, that can help determine human behaviors such as which crops to plant based on local climate conditions. Although enormous computational power can be applied to run the simulations themselves, *storing* the data that is generated during these simulations, and later *querying* it during subsequent offline analysis, can be a major challenge, especially with the trend toward very high resolution simulations. In many cases, the inability to offload the data onto a storage device in a timely manner leads domain scientists to throw away much of the data that is generated, maybe by sampling at a lower resolution, or by storing only a subset of the simulation variables, or by summarizing in various ways. Evaluating long-term importance of any specific data products is difficult at simulation time, thereby rendering these options unattractive, and often simulations need to be re-run when deficiencies in the stored data are revealed. To give an idea of the amount of data generated, a single day of simulated time for mesoscale climate modeling can generate 30 GB of data, while an entire ensemble simulation requiring the equivalent of several thousand years of simulated time can generate more that 30 PB of data [13].

Most of the work so far has focused on how to analyze or process the data once it has been stored in a scientific database management system. Some related work in this domain includes SciDB [26], which employs well-known compression techniques to reduce the storage footprint of the stored data; Seering et al. [22] address the problem of storing versions of an array and efficiently retrieving a specified version or versions. Recent work by Soroush et al. [24] and Bicer et al. [3] address the issue of compressing scientific simulation datasets, by relying on a preselected compression technique that may not provide the best compression ratio for every dataset. Overall prior techniques are not aggressive enough to deal with the high volumes of diverse data produced in many scientific disciplines such as climate modeling or astro-physical simulations.

To address the challenge of reducing the data volume after it has been generated but before storing it, we can use sophisticated compression techniques that exploit the high spatial and temporal correlation that exists in the data generated by scientific simulations. Among all the compression techniques, arithmetic encoding is a basic compression scheme that can optimally exploit known data distribution information, as it can compress the data at the information entropy rate [28]. However arithmetic encoding is not as efficient as conventional compression techniques such as `zlib` [29] and `lzo` [15] when measuring throughput, so is seldom employed in situations where compression speed is important. While the conventional compression techniques are efficient for compressing the data, the overall compression ratio may be enhanced by using them in conjunction with some preprocessing techniques, such as computing deltas, which makes the data amenable to better compression. Further, it is essential to select the best compression procedure given a particular dataset, instead of using a fixed, preselected compression technique. While better compression ratios reduce space on disk, they may result in higher compression and/or decompression times. For example, `zlib` does a very good job of compressing a dataset, however it loses out to `lzo` in terms of compression/decompression speed. Therefore, it is essential to provide users with the flexibility of choosing the appropriate compression technique(s) based on their individual requirements.

Secondly, since floating point data is the most prevalent type of data generated by scientific simulations, it is important to consider compression techniques tailored for such data due to its high de-

[*]Submitted towards the fulfillment of requirement of a Scholarly Paper for Master of Science without thesis option.

gree of entropy, especially in the lower order bytes of the mantissa (fraction) part [21]. Moreover, the full precision of floating point data may not always be required for some applications, e.g., visualizations, where truncation of the lower order mantissa bits can be tolerated. As a result, we may not be required to retrieve all the bytes of floating point data for query processing [11]. Therefore we can perform a byte-wise partitioning of the floating point data before storing on disk. The compression and the partitioning techniques discussed so far have been proposed independently in prior work, but have been mainly applied to small datasets.

In this paper, we present a comprehensive framework called PStore that is aimed at addressing the challenges in storing and managing high-resolution scientific simulation data. The data ingestion module of PStore has routines for preprocessing the data along with the backend compression routines; it also contains an analyzer that decides the appropriate preprocessing and backend compression modules to use for a specific scenario. In addition to compressing the data, the ability to query the stored data for offline analysis is also required. We currently support range queries in the query processing module of PStore. To reduce query latencies, chunking the data is essential, as it helps to alleviate dimension dependency [19]. PStore supports both single-level chunking (normal chunking) and two-level chunking [23, 25], while taking into consideration the effects of compressing the data. For expensive range queries, PStore also supports query prefetching to hide I/O latencies by overlapping I/O with the CPU processing time. Finally, we parallelize the query processing framework to further reduce overall query processing time.

This paper makes the following contributions:

1. Our framework supports a suite of compression schemes and can select the best compression plan based on the nature of the data. This procedure is carried out in an offline mode where a representative sample of the data selected for ingestion is analyzed and a compression plan is selected.

2. We integrate both byte-wise partitioning of floating point data and partitioning along data array dimensions to provide maximum flexibility in terms of accessing the desired data elements. Partitioning the floating point data elements bytewise involves partitioning the data across byte boundaries, which improves the compressibility of the data [21].

3. We propose a two-level partitioning/chunking strategy in the context of compression and show that it is better off (w.r.t the query response time) compared to a single-level chunking with compression.

4. For long running range queries, where several data chunks are accessed, it is beneficial to hide I/O latencies by overlapping them with CPU processing time. Since file accesses are often sequential, we can process one chunk at a time and overlap the I/O access for the next chunk with CPU processing of the current chunk.

5. We have built PStore, an end-to-end framework that is capable of ingesting high volumes of data from scientific simulations as it is produced.

The rest of the paper is organized as follows: The next section provides an overview of PStore. Section 3 gives a detailed description of the compression schemes that can be used in the framework. The two data partitioning techniques are described in Section 4, followed by the query processing techniques in Section 5. Experimental results are reported in Section 6 while Section 7 covers additional related work. Finally, Section 8 concludes the paper.

2. STORAGE FRAMEWORK

We present a short overview of PStore. PStore is an end-to-end framework containing two modules, namely, 1) a data ingestion module and 2) a query processing module. The data ingestion module can execute in parallel with the simulation that is generating the data, and performs both partitioning and compression of the data. The data generated by the simulation is transferred to the data ingestion module one *snapshot* at a time, as the snapshots are progressively generated (a snapshot typically corresponds to a time step in a physical simulation). The data ingestion module sends the compressed data chunks to the storage system as they are produced, and may have to wait to gather enough data before compression. The compressed chunks are also inserted into an index structure such as an R-tree [8], for querying the data efficiently. The query processing module runs offline in a separate computing environment decoupled from the data ingestion module.

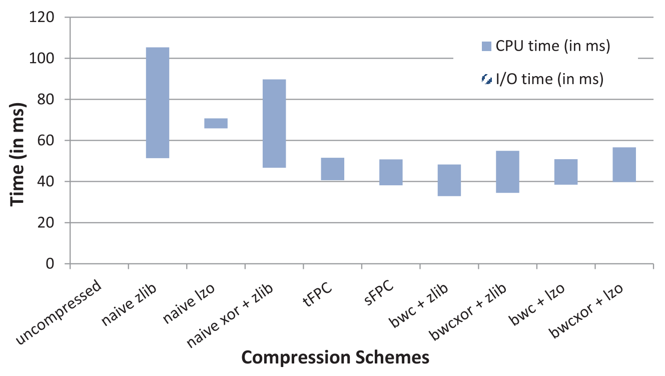
The data ingestion module begins execution by analyzing a sample of the data to be stored in an offline mode and chooses the appropriate compression scheme suited for the given dataset. Thereafter the multi-dimensional simulation data is partitioned along three different dimensions: (1) temporal, (2) spatial, and (3) bytewise. Partitioning the multidimensional data along both temporal and spatial dimensions is a well-known technique for alleviating dimension dependency [19]. In addition to single-level array partitioning, we implement a two-level array partitioning technique [18, 23, 25] and propose a variant of the technique when the effect of compression is taken into consideration. Another dimension of partitioning is bytewise, motivated by the observation that the full precision of the data is not required in many offline analysis and visualization tasks. Therefore the data is partitioned along byte boundaries, with the added benefit that the compression *efficiency* for this storage strategy is significantly better than traditional compression techniques for floating point data [21].

The PStore query processing module supports array slicing queries (or range queries). For retrieving the data given a range selection query, PStore submits the query to the R-tree to get the set of chunks to be retrieved from the disk. We perform two-level chunking with compression, in an attempt to read the minimum amount of data from the disk. As we will see later, this technique also enables us to speed up the decompression process during data retrieval. To speed up the query processing further, we employ a prefetching technique that hides disk access latencies behind decompression computations during data retrieval.

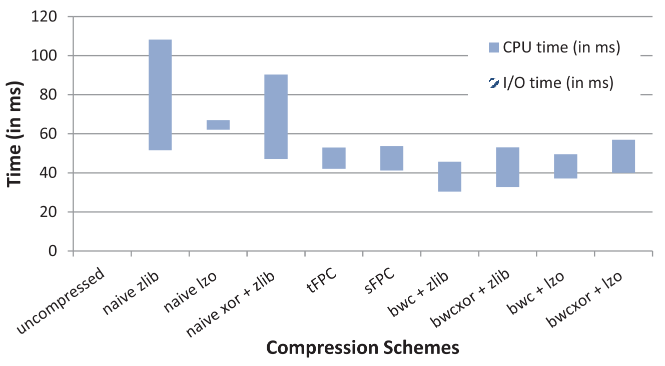
3. COMPRESSION SCHEMES

We describe the different compression schemes supported by our framework. Unlike most prior work (e.g.,[24]), we do not rely on a single compression scheme, and instead determine the compression scheme based on the structure of the data, or the requirements of the user. The data to be compressed is analyzed in an offline mode to determine the best scheme, in accordance with the user preference or the overall efficiency (based on both compression ratio and compression/decompression time). The simulation data that is currently compressed is analyzed periodically to ensure that we are still using the best compression scheme even when the distribution of the simulation data changes. We now discuss the different compression schemes implemented in PStore.

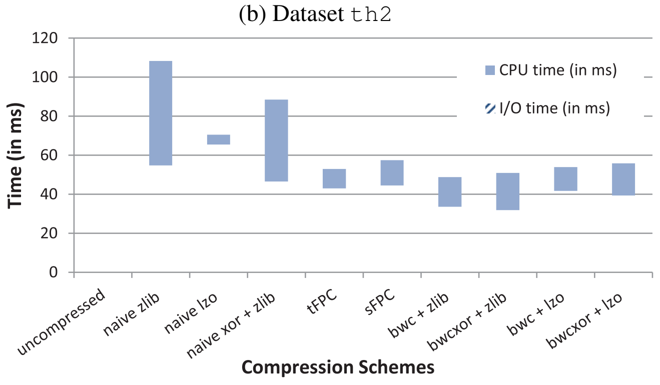
Bytewise Compression (bwc): This compression technique is similar to the one described by Schendel et al. in a recent work [21]. Data is partitioned into columns of bytes and the compressible bytes are identified by computing the byte value frequency distribution. The compressible bytes are then compressed using a backend compression algorithm such as `zlib` [29] or `lzo` [15]. We use `zlib` to obtain better compression ratios, and `lzo` to achieve faster compression and decompression rates as required. However unlike in [21], we do not compress and store all the compressible bytes together, as that adds considerable overhead to reconstruct the data due to required reshuffling of the bytes. Further, for approximate query processing where only a contiguous subset of the bytes are required, it is wasteful to decompress those bytes that are not required for answering the given query. As an exception to this strategy, only the two most significant bytes of a variable are stored together regardless of whether both are compressible or not. The reasoning behind this exception will be discussed in Section 4.2.



(a) Dataset `t2m`



(b) Dataset `th2`



(c) Dataset `psfc`

Figure 1: Total execution time for data retrieval with different compression schemes, for three datasets. The datasets are described in Section 6.

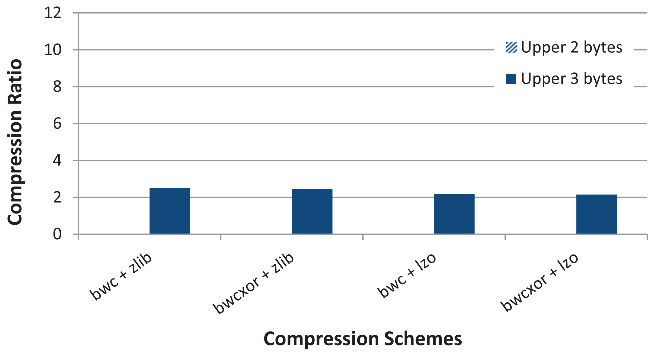
Bytewise-XOR Compression (bwcXOR): For scientific simulation datasets, there is a high degree of spatial and/or temporal correlation between neighboring array elements. As a result, the magnitude of the difference between two adjacent spatial or temporal data elements in the dataset may be small. Determining such correlations is essentially a preprocessing step towards better compressibility. The method for computing the difference between variables that we choose is XOR, rather than subtraction, as it yields a higher compression ratio both with two's complement integer and sign-magnitude floating point number representations [4]. After the data is partitioned into columns of bytes and the compressible byte columns are identified, we apply the XOR operation between the bytes of two different variables located spatially or temporally adjacent to each other. However, the XOR operation is not applied between incompressible byte columns as those are highly entropic and the resultant XOR value between two entropic bytes is also highly entropic. This is especially true for lower order bytes in a floating point variable where the bit randomness is often very high. The XORed byte columns are then compressed using one of the backend compression algorithms. All the snapshots except the last snapshot in a given data chunk are XORed in a temporal XOR operation. A spatial XOR operation can be applied on the last snapshot to further enhance the overall compression or it may be left unaltered, depending on the degree of spatial correlation between the variables and/or the amount of compression/throughput required.

Compression Method	**`t2m`**		**`th2`**		**`psfc`**	
	CR	% gain	CR	% gain	CR	% gain
naive `zlib`	1.325	0.00	1.305	0.00	1.279	0.00
naive `lzo`	0.996	-24.83	0.996	-23.68	0.996	-22.12
`naivexor+zlib`	1.375	3.77	1.371	5.05	1.474	15.25
tFPC	1.510	13.96	1.509	15.63	1.682	31.51
sFPC	1.615	21.89	1.631	24.98	1.529	19.55
`bwc+zlib`	**1.823**	**37.58**	**1.833**	**40.46**	1.823	42.53
`bwcxor+zlib`	1.798	35.70	1.804	38.24	**1.880**	**46.99**
`bwc+lzo`	1.686	27.25	1.696	29.96	1.643	28.46
`bwcxor+lzo`	1.669	25.96	1.676	28.43	1.776	38.86

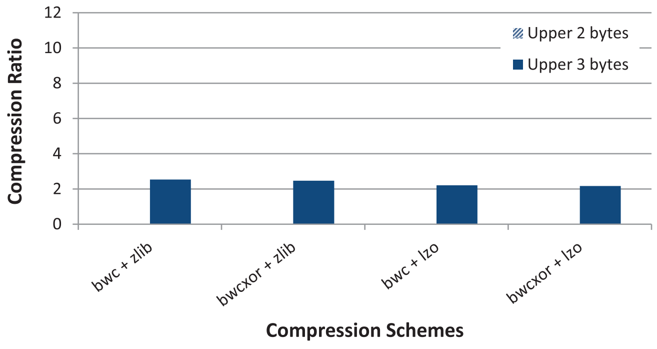
Table 1: Performance comparison of compression ratios between different compression schemes (compression ratio (CR) and % improvement relative to `zlib` (% gain) for each dataset). The best scheme for each dataset is highlighted.

This snapshot is used for retrieving the prior snapshots by applying the XOR operation in the reverse direction, a process referred to as *unrolling* hereafter. For retrieving the first snapshot, all prior snapshots needs to unrolled first and hence this technique comes with an overhead due to this unrolling process. We have designed this compression technique as an alternate compression scheme compared to the ISOBAR technique [21].

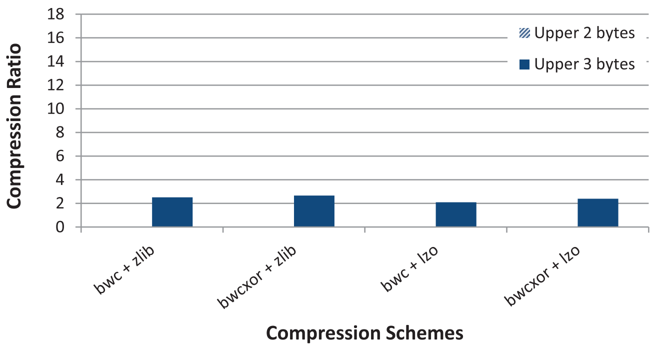
FPC: FPC is a compression technique developed for compressing 64-bit floating point data [4]. FPC predicts values by sequentially using two predictors (`fcm` [20] and `dfcm` [7]) and selects the value closer to the actual value. Thereafter, FPC performs an XOR between the two values and encodes the leading zero bytes of the result using three bits. The scheme uses an additional bit to specify which of the two predictors was used for prediction. The resulting 4-bit code and the non-zero residual bytes are written to the output. We observe that in the context of climate or simulation datasets, the value from the predictor in FPC can be replaced with an adjacent spatial or temporal value (due to the high degree of correlation between neighboring elements in these datasets). The removal of the predictor from the FPC algorithm speeds up its execution. In addition, the single bit of storage which is needed by the predictor is no longer required. We also extend FPC for single precision floating point data as well. For 32-bit floating point values, we assign two bits for counting the leading zero bytes, although there are five different possibilities (0 to 4). In the context of our climate datasets, we have observed that the count of four leading zeroes occurs least frequently. As a result, all XOR results with four leading zero bytes are treated the same as values with only three leading zero bytes, with the fourth zero byte emitted as part of the output. Our framework supports two different versions of FPC, sFPC and tFPC to denote XORing along spatial and temporal dimensions, respectively.



(a) Dataset `t2m`



(b) Dataset `th2`



(c) Dataset `psfc`

Figure 2: Compression Ratio for different values of byte-precision for single precision datasets

Other schemes: In addition to the compression schemes described above, our framework also implements the naive compression algorithms that apply `zlib` or `lzo` over the data. As an alternative to this approach, an XOR of the variables along the spatial or temporal dimensions can be performed followed by the application of one of the backend compression algorithms.

Experimental comparison and discussion: Table 1 presents compression ratios and their percentage improvement over the naive approach for three different datasets `t2m`, `th2` and `psfc`. The data is obtained from a simulation of the CWRF climate model [5]. A detailed description of the data is provided in Section 6.

First, we consider the use of a difference operator (i.e., XOR) as a pre-processing step. The use of the difference operator between correlated data values for better compressibility has been advocated in [24] for compressing snapshots in a scientific database system. They propose the use of variable-length delta encoding and subsequently using run-length encoding for compressing the bitmasks. The work targets compressing large number of zeroes and the small magnitude differences generated in the process of delta encoding. However, the technique may not be suitable for compressing floating point values, since taking the differences between two floating point values does not always result in small bit differences in values due to the way in which floating point numbers are actually represented. Further, the number of zeroes after delta encoding is comparatively small. We must keep in mind that a high degree of correlation between adjacent variables does not necessarily result in a zero in most cases for floating point numbers, due to the highly entropic low-order mantissa bits. We note from Table 1 that even selectively applying a difference operator between compressible bytes (*bwcXOR*) does not always turn out to be profitable when measuring compression ratio. The compression ratios of *bwc* for the datasets `t2m` and `th2` are better than *bwcXOR*. In addition, the decompression cost for schemes employing a difference operator for enhancing compressibility is higher than those without them. However, we observe that *bwcXOR* outperforms *bwc* for the dataset `psfc` and therefore justifies the inclusion of *bwcXOR* in the suite of compression schemes in the framework. Among all the compression schemes, the *bwc* schemes with or without the difference operator turn out to be the best for floating point data when measuring compression ratios. Our primary intent in this analysis is to establish that the use of the difference operator in scientific datasets, especially for datasets having a high degree of correlation, may not always turn out to be beneficial in terms of improving the compression ratio. Therefore care must to taken to choose the compression scheme selectively, rather than relying on a single compression scheme.

Better compression ratios for the data especially in a setting that involves huge amounts of simulation data being generated is an absolute necessity. In addition to reducing storage requirements, better compression also reduces the bandwidth requirement for transmitting the data to storage, or put another way, enables the data to be delivered faster from the generation site to the storage nodes. However, the stored data needs to be analyzed (or queried) later, hence efficient retrieval of the data is as important as developing better compression techniques for storage efficiency. Therefore we also need to incorporate query response time while choosing a compression technique for a given dataset. In Figure 1, we show the total execution time, which includes the I/O time (time taken to retrieve the data from the disk) and the CPU time (which includes the time to decompress the data, unroll the data when we perform an XOR operation and reshuffle the data for the *bwc*) for different compression schemes that PStore supports. We also demonstrate the I/O time for uncompressed data; the CPU time is zero in this case as we do not compress or perform an XOR on it. Since compression efficiency is dependent on the input data size, usually with larger data chunks resulting in better compression ratios, we perform the experiments on a data chunk size of around 3 MB. We determined this value empirically by experimenting with the same datasets that we use in the current experiment and observe the 3 MB value to be similar to that determined in previous studies [21, 10, 27]. We observe that disk I/O time constitutes a small percentage of the overall execution time and as a result compression ratio plays only a small role in reducing the overhead of data retrieval. Instead we pay a high price for decompressing the data and therefore care should be taken in choosing the appropriate compression scheme if compression ratio is not the only priority. We note that for `t2m` the disk space savings due to *bwc* + `zlib` (compression scheme with the highest compression ratio) is 45.15% compared to 40.69% when we use `lzo` in combination with *bwc* while the latter scheme performs decompression faster. Therefore an application that does not desire much space savings but does require fast query response time might want to select the latter scheme. The best compression scheme for the `psfc` dataset results in 46.81% disk savings compared to 39.14% when *bwc*+`lzo` is used. However the overall execution time for *bwcXOR*+`lzo` is around 2.5× higher than that of the other scheme due to cost of the unrolling operation. This implies that although using XOR may lead to higher compression ratios, there is a heavy price to pay when querying data compressed by this scheme. We emphasize that although `lzo` may not be comparable to `zlib` in terms of compression efficiency, it still proves to be a useful backend compression scheme when query throughput is important.

4. DATA PARTITIONING

We describe the two different data partitioning techniques that are employed in our framework. We first describe partitioning along dimensions (which include both spatial and temporal dimensions) followed by bytewise partitioning. The former partitioning technique alleviates dimension dependency whereas the latter is useful for achieving better compression ratios and for answering certain types of queries.

4.1 Partitioning along Dimensions

The multidimensional data is partitioned (or *chunked*) regularly across both temporal and spatial dimensions, where all partitions are assigned an almost equal number of elements. Regular chunking of multidimensional data has been shown to be an effective partitioning technique for many types of array operations [25]. If available, we use workload information to choose the chunk size and shape; such workload-aware chunking can lead to significant speedups for range queries [19, 6, 17]. Sarawagi et al. [19] showed that the average number of block fetches for a given access pattern can be minimized by choosing the shape of a chunk appropriately.

A block B is defined as the unit of transfer used by the file system for data movement to and from the storage device. The shape of a chunk is specified by the tuple $(c_1, c_2, \ldots, c_n)$, where c_i is the length of the ith dimension of the multidimensional chunk. A probability is assigned to each query access pattern independent of the actual position of occurrence in the array and the positions are assumed to be uniformly distributed across the entire domain. Therefore access patterns can be represented as $\{(P_i, s_{i1}, s_{i2}, \ldots, s_{in}) : 1 \leq i \leq k\}$ where k is the number of different classes of queries and P_i is the probability of occurrence of the ith class. Queries in this case are specified by an n-dimensional hypercube with only lengths of accesses in each dimension. The problem with the formulation for average number of block fetches by Sarawagi et al. [19] is that in some query instances the computed number of blocks to be fetched is exactly one less than the actual number of blocks to be fetched. That error is amplified due to the multiplication of factors across dimensions. Thus the error is significant if it is made for the majority of the dimensions of a given class. Otoo et al. [17] therefore modify the objective function as follows:

$$\sum_{i=1}^{k} P_i \prod_{j=1}^{n} \left(\frac{s_{ij} - 1}{c_j} + 1 \right) \tag{1}$$

In order to minimize the amount of additional data fetched from the disk, the chunk shape must satisfy the constraint:

$$\prod_{i=1}^{n} c_i \leq B \tag{2}$$

The goal in that prior work was to choose the chunk shape satisfying Eq. 2 that minimizes Eq. 1.

However, we note that the effect of compression has not been taken into account in earlier work, in computing the optimal chunk shape for array storage. We want to compress the data before it is stored in secondary storage to reduce the storage footprint on disk and also to maximize the disk bandwidth utilization. If the data has to be transferred over a network to/from the storage nodes, compression helps to reduce the transfer time as well. The compression ratio of standard compression algorithms like `zlib` also starts to degrade if the chunks contain too few bytes; let us denote such a threshold by B_c. This threshold may be different for different types of data and different algorithms, but can be easily learned given a sample dataset and an algorithm. Above this threshold, the compression ratio usually stabilizes to a fixed ratio ρ. To guarantee a good compression ratio, we place the following constraint on the chunk shape:

$$\prod_{i=1}^{n} c_i \geq B_c \tag{3}$$

At the same time, we want the compressed chunk to fit within a multiple of the disk block size B:

$$\frac{\prod_{i=1}^{n} c_i}{\rho} \leq mB \tag{4}$$

The threshold B_c is usually a multiple of the block size B and thus $B_c \leq m'B$ where m, m' are positive integers. We then have the following relation:

$$m'B \leq \prod_{i=1}^{n} c_i \leq m\rho B \tag{5}$$

where $m' \leq m$. Our intent here is to show how the constraints can be modified to incorporate the effect of compression into the optimization process. It is then not difficult to compute the optimal dimensions using the procedure outlined in [17].

The bytewise precision partitioning technique results in a compressed and an uncompressed data chunk corresponding to the compressible and incompressible bytes in a floating point data set. For best performance, this implies that there should be two different chunking strategies, one for the compressed bytes and another for the uncompressed ones, tuned according to the query workload. However, supporting two different chunking strategies would require maintaining two separate index structures which might prove to be a costly overhead. Another problem with this approach is that it requires two separate chunking strategies to chunk two different data representations. As a result, the number of snapshots in a chunk for the compressed data may be different from that required for the uncompressed data, which would require buffering and could slow down the overall chunking process. For these reasons, we do not chunk the data differently in the current implementation, nevertheless it would be an interesting study for future work to compare performance between these two alternatives.

We must keep in mind that with larger chunk sizes there is a higher price for decompression. Instead we can extend the idea of a *two-level chunking scheme* [25] by not compressing the entire chunk, but first partitioning into sub-chunks and then compressing the sub-chunks separately. We write the entire chunk to the disk. With this design, we do not have to decompress the entire chunk but only those sub-chunks that are required to answer the query. With this design, we just need to ensure that the sub-chunks satisfy Eq. 3 and the whole chunk satisfies Eq. 4, maximizing both compression ratio and I/O performance.

4.2 Bytewise Partitioning

In many applications, the full precision of the data may not be needed. For example, the lower order mantissa bits of a floating point number may be truncated during some types of visualization applications, since the human eye may not be capable of perceiving such fine differences. The IEEE 754 standard [12] for floating point arithmetic represents single precision values as a single sign bit, 8 exponent bits and 23 mantissa bits. Representing double precision values requires a single sign bit, 11 exponent bits and 52 mantissa bits. The mantissa bits in a floating point number represent a fractional component in the overall value and the lower order bits each contribute an exponentially smaller value to the overall magnitude of the number as we move from the higher to the lower order bits. This implies that discarding the lower order bytes in a floating point number (where the mantissa is stored) introduces less error compared to discarding higher order bytes. In contrast, truncation by discarding lower order bytes is not a feasible option for integer data due to the loss of significant bits that is not mitigated by multiplication by an exponent as for floating point data [11].

Table 2 presents the maximum relative error produced based on the number of mantissa bits retained for both single and double precision floating point data. Due to the small maximum relative errors introduced due to truncation of mantissa bits, it might suffice for applications to only retrieve the higher order k bytes corresponding to the amount of error the application can tolerate. This format of data access also requires partitioning the values in bytewise fashion. Therefore bytewise partitioning of values serves the dual purpose of enabling precision level partitioning and enhancing the compressibility of the data. While partitioning the data bytewise we always store the higher order two bytes together as they contain the sign and the exponent bits and truncation of exponent bits would introduce unacceptably high error rates. Also due to the expected high degree of correlation in the data, the higher order bits (which include the sign, exponent and higher order mantissa bits) of adjacent variables are likely to be similar. As the exponent bits in both single and double precision numbers span the first two higher order bytes, it is beneficial to store them together to enhance the compressibility of the data. However, partitioning the numbers bytewise requires reconstructing them when retrieving the data, which will introduce some overhead. When partial precision data is retrieved, the missing bytes are replaced with a fixed pattern as defined by the user.

5. QUERY PROCESSING

We support range queries in the query processing module of PStore. For retrieving the data for a range selection query, PStore determines the overlapping chunks in the query range and locates the data chunks on disk using the R-tree index. After the chunks are retrieved from disk, overlap with the sub-chunks within every chunk is determined from the query region. The overlapping sub-chunks are then extracted by decompressing them (if they were compressed in the data ingestion state) after determining their location from the header information that was stored along with the each chunk.

Decompression is an expensive operation and results in CPU processing time that can be even more than the time for I/O operations. We reduce this overhead by parallelizing the decompression. Sub-chunk decompression can be parallelized as every sub-chunk can be decompressed independently once a chunk is retrieved from disk. However, this process does not always lead to a linear speedup with the number of CPUs available. This is because not all the sub-chunks need to be extracted from a chunk, as they may not overlap with the query region. Further, since the process of sub-chunk extraction is overlapped with the chunk retrieval from disk (as described in Section 5.2), the overall processing time decreases with increasing parallelism until the CPU processing time becomes equal to the I/O processing time.

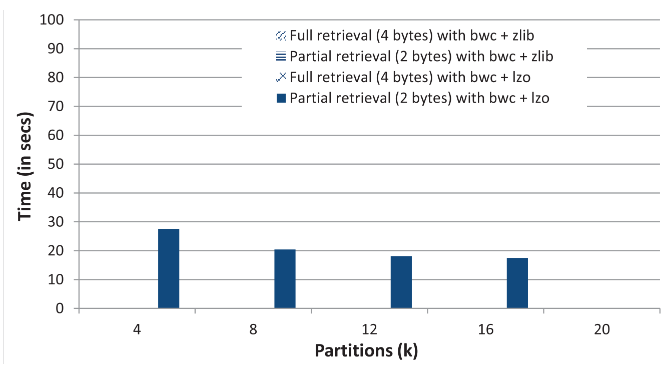
Significant Bytes	**Max. Error% (SP)**	**Max. Error% (DP)**
2	2.6e-1	3.1e0
3	1.0e-3	1.2e-2
4	-	4.8e-5
5	-	1.9e-7
6	-	7.3e-10
7	-	2.8e-12

Table 2: Maximum relative error due to reduced precision of IEEE 754 single and double precision floating point numbers

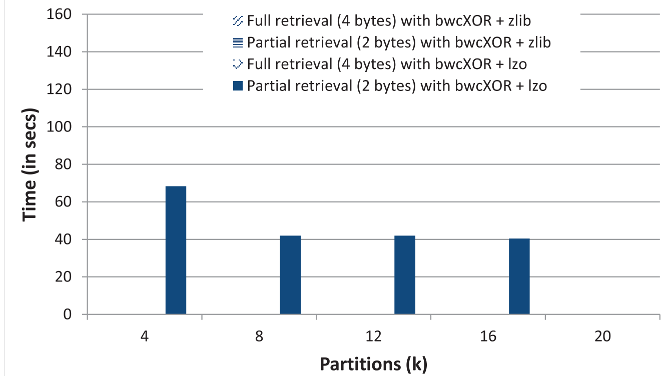
For approximate query processing, the application retrieves fewer bytes from disk, since the target application is able to tolerate lower precision. So the overall processing time should be lower compared to retrieving all the bytes for each data element from disk. Moreover, for partial precision data retrieval, usually the higher order bytes for the data elements are retrieved, which are generally more compressible than the lower order bytes. As a result the I/O operations are less expensive than when retrieving all (or only lower order) bytes of the data elements.

5.1 Two-level chunking with compression

Larger chunks help amortize disk seek overhead but pose a problem when the query region is a small subset of the chunk. We do not have random access to individual elements inside a chunk and therefore it is wasteful to process additional elements when the actual query region is a small contiguous fraction of the entire chunk. Smaller chunks however, reduce the average processing time for accessing an element in a chunk but introduces overheads from additional disk seeks. Two-level chunking seeks to balance the two factors. A larger chunk is split into regular sized sub-chunks so that the overall processing overhead is minimized. The larger chunks are the units of disk I/O while the smaller chunks form the unit of array processing. Two-level chunking has been studied previously [18, 23, 25], but without including the effects of compression. There are several strategies we may choose to apply when using two-level chunking with compression. One strategy is to compress the chunks before writing them to disk. The drawback of this approach is that in the case of range queries, it is seldom the case that



(a) Retrieval time using *bwc*+`zlib` and *bwc*+`lzo`



(b) Retrieval time using *bwcXOR*+`zlib` and *bwcXOR*+`lzo`

Figure 3: Partial (upper 2 bytes) retrieval time vs. full (4 bytes) retrieval time for different compression schemes for dataset `T`

one needs to access all the data elements contained in the larger chunk. As a result, we must pay the cost of decompression for the additional data that is not needed. As we observed previously, decompression is a relatively costly operation, therefore it might be beneficial to compress the smaller sub-chunks individually and then combine them into a chunk before writing to the disk. Two-level chunking also opens up parallelization opportunities by allowing us to process the sub-chunks in parallel and thus speed up query processing. In our implementation, we follow a regular chunking scheme for both chunks and sub-chunks, as this chunking strategy has been shown to yield the best performance compared to irregular chunking schemes [25] and determine the optimal chunking layout empirically.

5.2 Chunk prefetching

For long running range queries, where multiple chunks may be accessed, we can improve performance by hiding I/O latencies by overlapping them with CPU processing time. Since a chunk access is sequential, we want to process one chunk at a time and overlap the I/O access of the next chunk with CPU processing of the current chunk. With two-level chunking, we can further process the sub-chunks in an embarrassingly parallel fashion as all the computations related to each sub-chunk can be executed independently. This is advantageous because we require large chunks to amortize the I/O time and simultaneously the presence of a large number of sub-chunks in a chunk enables higher throughput.

6. EXPERIMENTAL RESULTS

We evaluate the performance of the different components that constitute PStore. We use a real dataset for this purpose which is based on the Regional Earth System Model (RESM) to provide climate and environmental information for a wide range of end users with drastically different data demands. RESM is based on the state-of-the-art regional climate model CWRF [5] that predicts mesoscale climate processes, including atmosphere, hydrology, crop, soil, air and water quality and their interactions. The dataset contains numerous variables each of which records measurements of parameters such as temperature, pressure, relative humidity, wind velocity, actual biogas emissions, CO concentration, etc. The measurement is either performed on a region of space defined by a 2D grid or a 3D grid and is recorded periodically over a fixed time duration. Each such grid is termed a *snapshot* at a particular instance in time. We perform our evaluation on both 3D and 4D datasets, which are described below.

3D dataset: Each snapshot in this dataset is a 138×195 array recorded over a 3 hour interval, so there are 8 snapshots per day. Currently, we have simulation data for one month, which has a total 240 snapshots for all the variables. We used three variables (i) `t2m`: air temperature at 2m, (ii) `th2`: potential temperature at 2m, and (iii) `psfc`: surface pressure, for the purpose of experimentation with 3D datasets.

4D dataset: This dataset differs from the 3D dataset in that it has an added *height* dimension, which makes each snapshot a $138 \times 195 \times 35$ matrix. We used two variables from this 4D dataset `T` and `P` which denotes perturbation temperature and perturbation pressure, respectively.

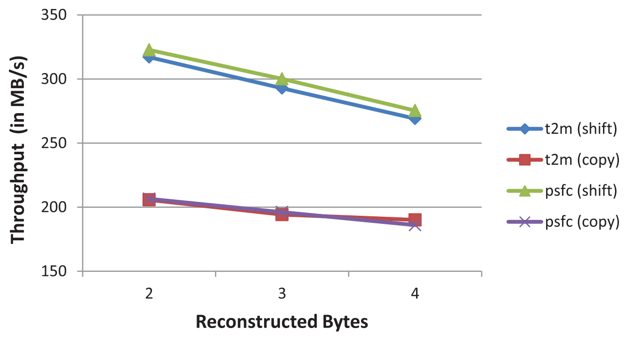
The total size of the dataset is around 140 GB and each measurement is stored as a single-precision floating point variable. The dataset is represented in the netCDF format [16].

We performed our experiments on an Intel Xeon® E7450 (2.4 GHz). This machine has 4 sockets each having 6 cores with a total of 24 threads and a 48 GB main memory.

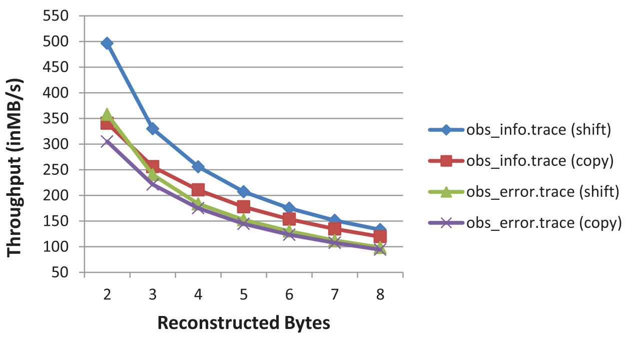
6.1 Effect of Bytewise Partitioning on Compression

We study the effect of bytewise partitioning on the compression ratio. While reading partial or reduced precision data from the disk, we always read the higher order bytes which contain the sign, exponent and a subset of the mantissa bits. For single precision data, we have two possibilities for reduced precision data, either 16 bits (upper 2 bytes) or 24 bits (upper 3 bytes). From Figure 2 we observe that for every compression scheme, the compression ratio of the higher order 2 bytes is best. This is because there is a high degree of correlation between the adjacent variables in this dataset and this correlation is captured best by the sign, exponent and the higher order mantissa bits, which causes these values to be very similar or equal for much of the data. We also observe that the compression ratio for 16-bit precision is almost $5\times$ better than for 24-bit precision. The large difference can be explained by the fact that lower order mantissa bits are highly entropic, so are responsible for the decrease in compression ratio. Therefore, an application that can tolerate 0.26% relative error in precision (see Table 2) can achieve a large savings in disk space and faster query response times due to the high compressibility of the data at reduced precision.

Figure 3 shows query retrieval times for full and partial precision data for different partitioning parameters with different compression schemes. Figure 3a shows the retrieval times when the upper 2 bytes of data are compressed with *bwc*+`zlib` and *bwc*+`lzo`,



(a) Single precision dataset: `t2m` & `psfc`



(b) Double precision dataset: `obs_info.trace` & `obs_error.trace`

Figure 4: Throughput of `shift` and `copy` reconstruction technique for (a) single and (b) double precision datasets

and Figure 3b shows the retrieval times when XORing had been applied during compression and unrolling must be done during query retrieval. The lower 2 bytes are not compressed because they are highly entropic. We observe that the partial query retrieval for *bwc*+`zlib` takes around 70% of the full query retrieval time across all partition configurations whereas the partial query retrieval time is 50% of the full query retrieval time for *bwc*+`lzo`, since the decompression time for *bwc*+`zlib` is higher than for *bwc*+`lzo`. From Figure 2 we see that the compression ratio for `zlib` is higher than for `lzo`. Moreover `lzo` fails to compress the data when $k = 20$, i.e., when sub-chunks are smaller, and therefore we have no data points at that value of k. Figure 3b shows that using XOR and unrolling introduces significant overhead in the retrieval process, which causes the partial query retrieval time to increase to around 80% and 70% of the full query retrieval time for *bwcXOR*+`zlib` and *bwcXOR*+`lzo` schemes, respectively.

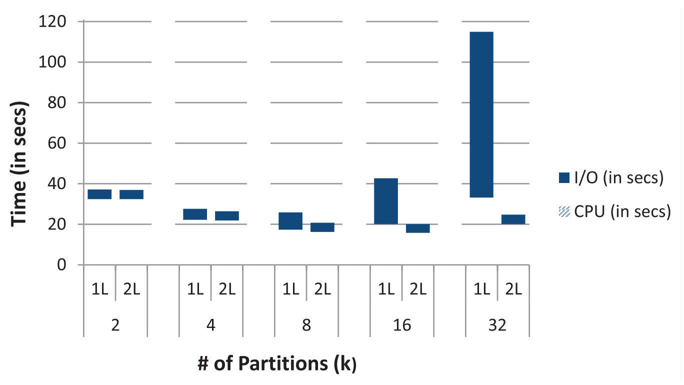
However, bytewise partitioning introduces some overhead due to the reconstruction required to build a floating point number from its individual bytes. We study the overhead for two different types of reconstruction techniques, measuring overall throughput. The first technique is to reconstruct the floating-point number by byte shifting while the second technique copies the individual bytes to their respective offsets in memory to reconstruct the original number. Figure 4 presents the throughput obtained while reconstructing the bytes using both the shift and the copy method. We observe that the shift method outperforms the copy method in the throughput obtained. This is because the copy method involves moving many single or small groups of bytes (if subsequent bytes in a variable are kept together in a partition) to the target location requiring multiple byte copy operations with each copy operation associated with some fixed overhead. In case of byte shift operation the entire data element (4 or 8 bytes) is constructed in-place by byte shifting (which is a cheap operation) and then assigned to the target variable requiring a single move operation. As previously noted, we reconstruct at least the most significant two bytes for both single and double precision data. The throughput gain is at least 54% and 17% when reconstructing the two most significant bytes, for single and double precision data respectively. We also note that, not surprisingly, the throughput gain decreases with an increase in the number of bytes reconstructed.

6.2 Two-level Chunking

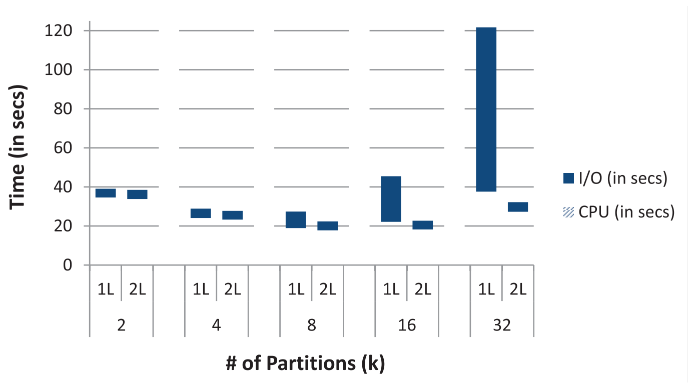
We demonstrate experimentally the variation in query response time as the number of sub-chunks and chunks for a given variable in the dataset is varied. We also show that our proposed approach to two-level chunking with compression outperforms single-level chunking with compression. The experiments are performed on both 3D and 4D datasets.

We compare single-level (1L) chunking to two-level (2L) chunking in the context of compression. We measure the performance of slicing queries for 1000 queries generated uniformly at random for a 3D dataset. For experiments involving 4D datasets, we restrict the number of such queries to 100, as this dataset is bigger than the 3D dataset by around an order of magnitude. The 3D dataset has two spatial dimensions while the 4D dataset has three of them, with each dataset having a temporal dimension denoting the timestamps in which the simulations were run. In the experiments, each spatial dimension is k-way partitioned, i.e. each dimension is divided into k partitions. Therefore each snapshot has k^d sub-chunks, where d is the number of spatial dimensions. For example, when $k = 4$ and $d = 2$, 16×16 snapshot is partitioned into 16 (4^2) sub-chunks, where each sub-chunk is stored in a separate file. The value t gives the number of temporal dimensions or snapshots that will be included in each file. If $t = 16$, then each file has $4 \times 4 \times 16$ elements, which we refer to as single-level chunking. This configuration can also be denoted by the tuple (k, t). For two-level chunking, we specify an additional parameter s, which denotes the number of chunks to partition the data into. In other words, it also indicates the number of sub-chunks that would be allowed in a chunk. The 16 sub-chunks that were created by 4-way partitioning before, can be viewed as a 4×4 array of sub-chunks. In a similar way, each dimension in this array is now s-way partitioned. For $s = 2$, the 4×4 array is partitioned into 4 (2×2) chunks and each chunk is stored in a separate file. We denote a two-level partitioning by the tuple (k, t, s). Although each spatial dimension is partitioned equally into k or s parts, our framework supports unequal partitioning across different dimensions as well.

Figure 5 shows a breakdown into CPU time and I/O time for the 3D datasets `t2m` and `psfc`. For 3D dataset, the chunk is always 2-way partitioned ($s = 2$) and the number of snapshots in a chunk is 16. From Figure 6, we observe that the performance of the 1L and 2L chunking strategies is similar for smaller number of partitions. Since the number of partitions is small, the number of chunks to be written to disk is small, resulting in fewer disk seeks. However the performance of the 1L strategy degrades with an increase in the number of partitions. This is because of the increase in disk seeks for the 1L strategy, whereas the number of disk seeks remains almost constant for all the configurations for the 2L strategy. This is due to the use of full chunks as the unit of disk access for the 2L strategy. The best performance for the set of partitions selected



(a) 3D dataset `t2m`



(b) 3D dataset `psfc`

Figure 5: Total execution time (CPU + I/O) for array slicing queries for single-level (1L) and two-level (2L) chunking with different partition numbers on two different datasets

is achieved for the (8, 16, 2) and (16, 16, 2) configurations for the 3D dataset, as can be observed from Figure 6. Increasing the number of sub-chunks beyond 16 decreases the performance of the 2L strategy. The decreased performance comes from an increase in CPU processing time, due to the overhead of decompressing a large number of relatively small sub-chunks.

We study the effect of variation in chunk size in Figure 7 and Figure 8 by fixing k and t. We observe that CPU time remains almost unaffected by the change in the chunking parameters, confirming that using chunk as the unit of disk I/O does not affect CPU time significantly. For the 3D dataset, $s = 2$ turns out to be the optimal choice of chunk partition.

This value of s is best for different values of k and t, as can be seen from Figure 7a and 7b. Figure 8a and 8b show the variation of s for the 4D dataset. In this case, $s = 5$ is the optimal choice for chunk partition. In general, the optimal value of s for a given dataset can be found by analyzing a sample of the dataset in the pre-processing step.

6.3 Parallelizing Query Retrieval

In a two-level chunking scheme, the sub-chunks can be processed in parallel once the chunks intersecting with the query region are retrieved from disk. However, these two processes are performed in a pipelined fashion using double buffering by default. To see this, we executed the query framework with partition parameters (8, 8, 4) and (8, 16, 4), since those configurations were the best parameters for the 4D dataset for the given query workload, determined empirically. We observe from Figure 9 that the framework does not scale beyond 8 cores/threads for these queries. As the number of core increases, the compute time decreases and becomes equal to the I/O time, which remains fixed irrespective of the increase in the number of cores. The application cannot perform better once the I/O time becomes greater than or equal to the compute time for the queries. For the parallelization process, we assign each compute thread a sub-chunk to perform the decompression process in parallel, to remove the performance bottleneck of the expensive decompression operations. However, we note that there are still load balance issues that limit performance even if we assign n threads evenly to the k^d sub-chunks ($n \leq k^d, d = 3$ in this case). It is likely that at some times in executing the queries, the number of sub-chunks to be processed will be less than n. This is because a query region might intersect with too few sub-chunks in some cases, or that different sub-chunks take different amounts of time to process. Load imbalance is therefore another reason why the application may not scale linearly with increasing number of threads, if overall performance is limited by CPU computations rather than I/O time.

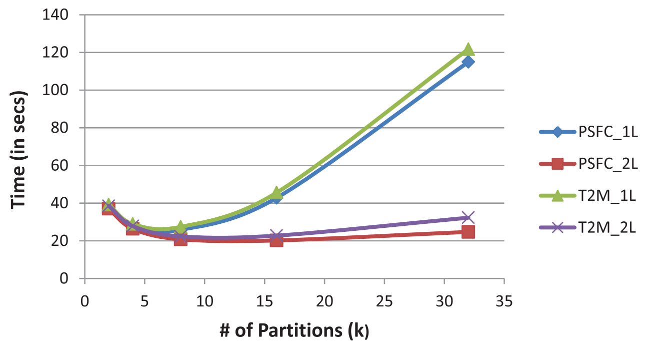
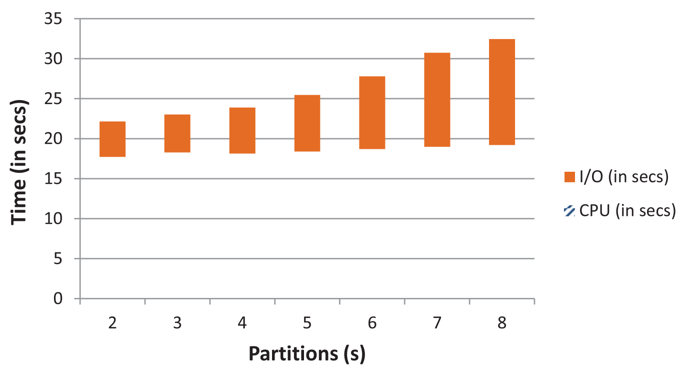


Figure 6: Performance of array slicing queries for single-level and two-level chunking.

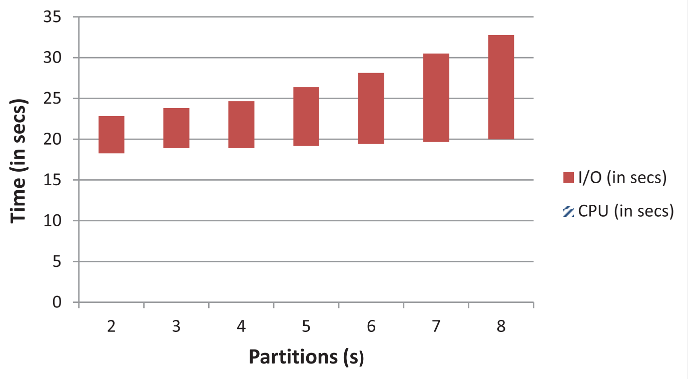
6.4 Chunk Prefetching

Figure 10 shows the reduction in query execution time due to chunk prefetching. Chunk prefetching is achieved by using double-buffering in memory with the chunk to be processed next prefetched and stored into memory while processing for the current chunk takes place from a different buffer. Additional buffers could be employed if more than one disk is available. To perform the prefetching, we use two separate types of threads; several dedicated solely to CPU processing (one per available CPU/core) while the other is dedicated to disk I/O. This straightforward optimization hides the disk access time behind the processing time, as long as the processing time dominates the I/O time, as we have seen is true in most cases we have experimented with. The current experiment is performed on the 4D dataset where the partition parameter k is varied from 4 to 20 in steps of 4, fixing t and s at 16 and 4, respectively. From Figure 10 we observe that it is possible to completely overlap the disk access time with the CPU processing time via chunk prefetching.

7. RELATED WORK



(a) Dataset `t2m`: $k = 8, t = 16$



(b) Dataset `t2m`: $k = 16, t = 16$

Figure 7: Query performance (CPU + I/O time) varying chunk size, for the 3D dataset `t2m`

Data compression has been very well studied and has been extensively used in databases and general storage systems to reduce storage requirements and improve query execution performance. More recently there has been a resurgence in interest in the storage and management of scientific data [2, 26, 25, 24, 22, 3]. Scientific observation and simulation data is typically stored in large arrays with snapshots stored in different array structures, as they are progressively generated. Due to the high degree of correlation that exists within and across the snapshots, delta encoding is commonly used to compressing such data. TimeArr [24] and Bicer et al. [3] use very similar delta encoding techniques for compressing scientific simulation datasets. Seering et al. [22] uses hybrid delta encoding where the data is first encoded using “dense” encoding technique and the result, which is relatively sparse so contains many zeroes, is encoded with a “sparse” encoding technique. We have seen that not all scientific datasets are amenable to delta compression and therefore there is a need to select the best possible compression scheme for a given dataset. Moreover, the delta encoding techniques we refer to do not consider floating point numbers explicitly, which is the most common type of data generated by scientific simulations. Due to the high degree of entropy present in the low-order mantissa bits, delta encoding between two floating point numbers does not result in low magnitude values which can then be stored using fewer of bits.

Schendel et al. [21] propose the ISOBAR system which does a byte-wise partitioning of floating point numbers and preprocesses the bytes to identify compressible and hard-to-compress bytes. They observe that it may not be effective to compress all the bytes as some of the bytes may be incompressible due to the high amount of randomness present in them. We extend their technique to improve the delta encoding performance for floating point numbers. This partitioning strategy comes with the added benefit of being able to query the data approximately. Not all applications require full precision data and if the user specifies a relative error bound, it is possible to avoid retrieving all the bytes from the disk which makes query processing more efficient [11].

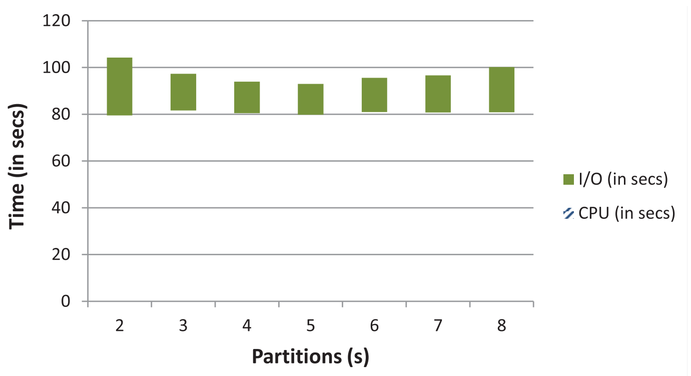
Other compression techniques for floating point numbers have been proposed, such as `fpzip` [14] and FPC [4] that are also used widely in scientific database applications. These techniques are based on context modeling applications, and use predictors for predicting the next value based on the values seen earlier in a sequence. The predicted value is then XORed with the actual value and the leading zero bytes are compressed. The performance of these techniques may degrade due to the use of predictors for predicting the next value. However for data generated by scientific simulations, one may do away with these predictors by using the next temporal or spatial value in an array since the values are highly correlated and are usually very close to the previous value. We extend FPC in our tFPC and sFPC compression methods, which essentially XOR neighboring temporal or spatial values, respectively.

Most of the prior work on storing and accessing multidimensional data does not support versioning explicitly. The array engines in use today, such as RasDaMan [1], are not specifically tuned towards supporting versioning techniques that are critical to no-overwrite storage systems. NetCDF [16] and HDF5 [9] are self-describing, machine independent data formats for array-oriented scientific data that physical scientists have long been using to store and access multidimensional data generated by scientific simulations. However these data formats do not support versioning either. Most of these systems do support chunking for alleviating dimension dependency.

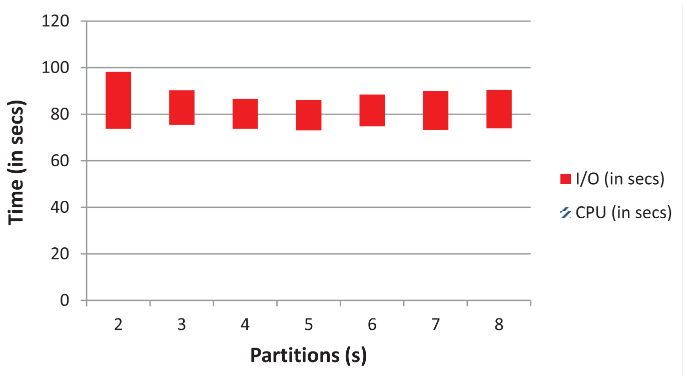
Prior work related to chunking multidimensional data has considered both single-level and two-level chunking for array storage [18, 23, 25]. In the two-level chunking scheme, the proposed techniques resort to different combinations of both regular and irregular tiling. However they do not consider the effect of compression while chunking the array. In our work, we propose a variant of two-level chunking that takes into consideration the effect of compression. The problem of tuning the chunk shape and size for a given query workload has also been considered previously [19, 17]. In our work, we show that the earlier formulations for computing the optimal chunk size can be modified to take into consideration the effect of compression.

8. CONCLUSION

In this paper, we presented the design, implementation, and evaluation of PStore, a no-overwrite storage manager that we are building for managing array data generated during scientific simulations. Unlike other scientific data management systems, PStore’s primary goal is to alleviate the data ingestion bottleneck, by compressing the simulation data as it is being generated and offloading it to storage nodes while minimizing the communication overhead. The data ingestion module in PStore contains a suite of compression techniques designed to handle diverse types of floating point datasets generated during scientific simulations, and we presented an approach to choose an appropriate compression technique based on the application needs. PStore also supports approximate query



(a) Dataset T: $k = 8$, $t = 16$



(b) Dataset T: $k = 16$, $t = 16$

Figure 8: Query performance (CPU + I/O time) varying chunk size for the 4D dataset T

processing by retrieving partial precision data if that is sufficient for the application needs, and contains several other optimizations for efficient query execution. Our extensive experimental evaluation illustrates that different compression techniques work better for different datasets, and further that using bytewise partitioning and two-level chunking can lead to significantly higher compression ratios and lower query execution times respectively.

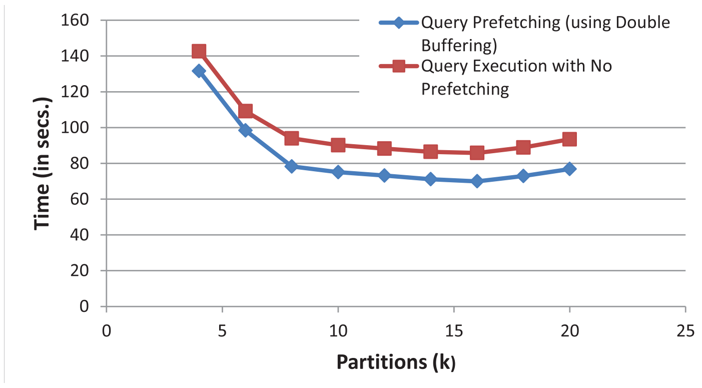


Figure 10: Effect of prefetching and double buffering on query performance with different partition configuration (k) on the 4D dataset P

9. REFERENCES

[1] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *SIGMOD Conference*, pages 575–577, 1998.

[2] P. Baumann, S. Feyzabadi, and C. Jucovschi. Putting pixels in place: A storage layout language for scientific data. In *ICDM Workshops*, pages 194–201, 2010.

[3] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1205–1216, 2013.

[4] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Computers*, 58(1):18–31, 2009.

[5] Climate-Weather Research and Forecasting model. http://cwrf.umd.edu/.

[6] P. Furtado and P. Baumann. Storage of multidimensional arrays based on arbitrary tiling. In *International Conference on Data Engineering (ICDE)*, pages 480–489, 1999.

[7] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 207–216, 2001.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[9] HDF5. http://www.hdfgroup.org/HDF5/.

[10] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *International Conference on Data Engineering (ICDE)*, pages 1199–1208, 2011.

[11] J. Jenkins, E. R. Schendel, S. Lakshminarasimhan, D. A. B. II, T. Rogers, S. Ethier, R. B. Ross, S. Klasky, and N. F. Samatova. Byte-precision level of detail processing for variable precision analytics. In *Supercomputing Conference (SC)*, pages 48–58, 2012.

[12] W. Kahan. IEEE standard 754 for binary floating-point arithmetic. http://www.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, October 1997.

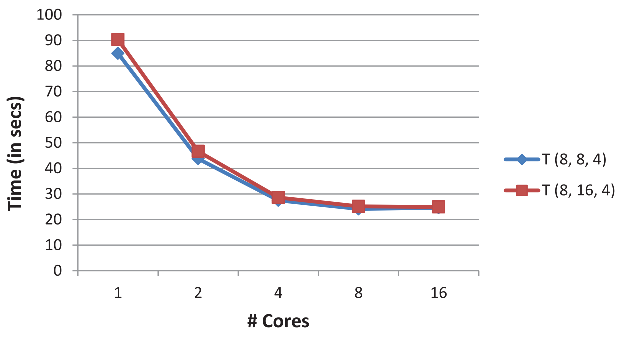
[13] X.-Z. Liang, M. Xu, X. Yuan, T. Ling, H. Choi, F. Zhang, L. Chen, S. Liu, S. Su, F. Qiao, J. Wang, K. Kunkel, E. J. W. Gao, V. Morris, T.-W. Yu, J. Dudhia, and J. Michalakes. Regional climate-weather research and forecasting model (CWRF). *Bull. Amer. Meteor. Soc.*, 2012.

[14] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1245–1250, 2006.

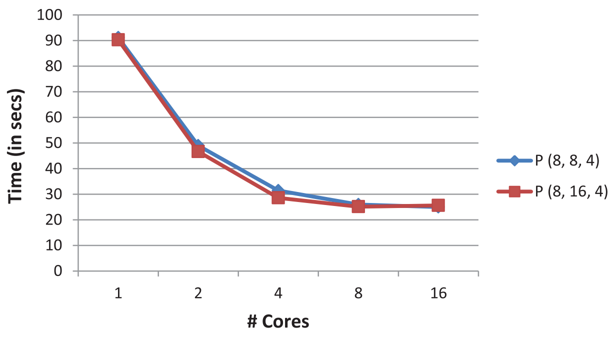
[15] Lzo compression library. http://www.oberhumer.com/opensource/lzo/.

[16] NetCDF: Network Common Data Form. http://www.unidata.ucar.edu/software/netcdf/.

[17] E. J. Otoo, D. Rotem, and S. Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *International Workshop On Data Warehousing and OLAP(DOLAP)*, pages 25–32, 2007.



(a) Dataset `T`



(b) Dataset `P`

Figure 9: Strong scalability of the query retrieval framework for 4D datasets for different partition configurations

[18] B. Reiner, K. Hahn, G. Hofling, and P. Baumann. Hierarchical storage support and management for largescale multidimensional array database management systems. In *International Conference on Database and Expert Systems Applications (DEXA)*. Springer-Verlag, 2002.

[19] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *International Conference on Data Engineering (ICDE)*, pages 328–336, 1994.

[20] Y. Sazeides and J. E. Smith. The predictability of data values. In *International Symposium on Microarchitecture (MICRO)*, pages 248–258, 1997.

[21] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.-S. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova. Isobar preconditioner for effective and high-throughput lossless data compression. In *International Conference on Data Engineering (ICDE)*, pages 138–149, 2012.

[22] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *International Conference on Data Engineering (ICDE)*, pages 1013–1024, 2012.

[23] T. Shimada, T. Tsuji, and K. Higuchi. A storage scheme for multidimensional data alleviating dimension dependency. In *International Conference on Digital Information Management (ICDIM)*, pages 662–668, 2008.

[24] E. Soroush and M. Balazinska. Time travel in a scientific array database. In *International Conference on Data Engineering (ICDE)*, pages 98–109, 2013.

[25] E. Soroush, M. Balazinska, and D. L. Wang. Arraystore: a storage manager for complex parallel array processing. In *SIGMOD Conference*, pages 253–264, 2011.

[26] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–16, 2011.

[27] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[28] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

[29] Zlib compression library. `http://www.zlib.net`.