ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying

John Jenkins[1,2,⋆], Isha Arkatkar[1,2,⋆], Sriram Lakshminarasimhan[1,2], David A. Boyuka II[1,2], Eric R. Schendel[1,2], Neil Shah[1,2], Stephane Ethier[3], Choong-Seock Chang[3], Jackie Chen[4], Hemanth Kolla[4], Scott Klasky[2], Robert Ross[5], and Nagiza F. Samatova[1,2,⋆⋆]

[1] North Carolina State University, NC 27695, USA
[2] Oak Ridge National Laboratory, TN 37831, USA
[3] Princeton Plasma Physics Laboratory, Princeton, NJ 08543, USA
[4] Sandia National Laboratory, Livermore, CA 94551, USA
[5] Argonne National Laboratory, Argonne, IL 60439, USA
samatova@csc.ncsu.edu

Abstract. High-performance computing architectures face nontrivial data processing challenges, as computational and I/O components further diverge in performance trajectories. For scientific data analysis in particular, methods based on generating heavyweight access acceleration structures, e.g. indexes, are becoming less feasible for ever-increasing dataset sizes. We present ALACRITY, demonstrating the effectiveness of a fused data and index encoding of scientific, floating-point data in generating lightweight data structures amenable to common types of queries used in scientific data analysis. We exploit the representation of floating-point values by extracting significant bytes, using the resulting unique values to bin the remaining data along fixed-precision boundaries. To optimize query processing, we use an inverted index, mapping each generated bin to a list of records contained within, allowing us to optimize query processing with attribute range constraints. Overall, the storage footprint for both index and data is shown to be below numerous configurations of bitmap indexing, while matching or outperforming query performance.

1 Introduction

Increasingly complex simulation models, capable of using high-end computing architectures, are being used to simulate dynamics of various scientific processes with a high degree of precision. However, coupled with this opportunity to augment knowledge and understanding of the highly complex processes being studied are the challenges of conducting exploratory data analysis and knowledge discovery. Specifically, data size on the tera- and peta-scale is becoming a limiting factor in understanding the phenomena latent in these datasets, especially in a post-processing context.

Due to massive dataset sizes, full context analysis is a crucial bottleneck in the knowledge discovery pipeline, being restrained by the limits of computer memory and

⋆ Authors contributed equally.
⋆⋆ Corresponding author.

I/O bandwidth. Most commonly, applications of which such data exploration processes are characteristic are interactive and require near-real-time I/O rates for full data exploration. However, I/O access rates are too slow to support efficient random disk access in real-time for large-scale data sets, necessitating new approaches to reduce the I/O pressure of extreme-scale data analytics.

A *knowledge priors* approach to data analytics is promising in restricting data to smaller and more practical sizes. Often times, scientists have some prior knowledge about the regions of interest in their data. For example, fusion scientists aiming to understand plasma turbulence might formulate analysis questions involving correlations of turbulence intensities in different radial zones ($0.1 < \psi < 0.15; 0.3 < \psi < 0.35; 0.5 < \psi < 0.55; 0.7 < \psi < 0.75; 0.9 < \psi < 0.95$). Likewise, climate scientists aiming to understand factors contributing to natural disasters might limit their search to particular regions or perhaps only a single region.

Thus, formulating queries on scientific simulation data constrained on variables of interest is an important way to select interesting or anomalous features from large-scale scientific datasets. Traditional database query semantics are an effective means to express such queries. This allows us to leverage a great deal of work from the database community on query processing. The indexing techniques used in traditional database systems, such as $B-$trees [9] and bitmap indexes [23], have been used extensively in the literature. However, while indexing is a blessing for fast and efficient query processing, it is arguably a curse in terms of storage; index sizes are often 100-300% of the original column size for high-cardinality data (such as double-precision data) [26], which is a huge bottleneck for storage- and I/O-bound extreme-scale applications.

A number of bitmap index compression techniques have been introduced to reduce the size of the bitmap index while maintaining fast query retrieval. In particular, Word Aligned Hybrid (WAH) [24] bitmap compression is used in FASTBIT [23], a state-of-the-art scientific indexing technology with fast query processing capabilities. Notably, however, the total storage footprint for a high-cardinality data column along with an associated FASTBIT index is around 200% of the original size [25], which is still prohibitive in many extreme-scale contexts. Furthermore, while this indexing scheme is optimized for region-retrieval queries over spatio-temporal data sets (i.e., returning the record IDs/regions that match a query constraint), returning the *actual values* of the variables associated with these regions (i.e. value retrieval query) is equally important in data analytics, necessitating an expanded approach.

Therefore, we present ALACRITY, an Analytics-driven Lossless Compression methodology, for Rapid in-situ Indexing, sToring, and querYing. ALACRITY integrates data reduction and indexing methodology for floating-point datasets, optimized for query-driven data analytics over scientific data. We believe that a tight cohesion between the data and index allows us to optimize storage requirements while at the same time facilitating both fast indexing at simulation-time and range query processing with value retrieval during analysis. In particular, our focus is on write-once, read-many (WORM) datasets utilizing double-precision floating-point variables, as are commonly produced by large-scale, high-fidelity simulation runs and subsequently analyzed by numerous application scientists in multiple (often global) contexts. A few examples of such data are the particle-based fusion simulation GTS [20] and the direct numerical combustion simulation S3D [8], each of which are comprised of primarily double-precision, high-cardinality variables ($\approx 100\%$ unique values for GTS, $\approx 50\%$ unique values for S3D).

Toward the goal of developing a system given this motivation, we make the following contributions in this paper:

– We present a lossless compression methodology for floating-point (single and double-precision) variables utilizing unique-value encoding of the most significant bytes. Our lossless compression reduces the size of a number of high-entropy, double-precision scientific datasets by at least 15%. Compared to lossless compression techniques like FPC [6], optimized for floating-point data, we report superior average compression ratios.
– Using our lossless compression method, we optimize range query evaluation including value retrieval by binning the column data by the distinct significant byte metadata, integrating efficient compressed-data organization and decompression of retrieved results. Compared to state-of-the-art techniques like FastBit [23], we provide comparable or better performance on range queries retrieving record IDs. For range queries that additionally retrieve variable values, we achieve up to one order of magnitude improvement in performance.
– For query processing, we utilize an inverted index, incurring a smaller storage footprint compared to other database indexing schemes. Using an inverted index compression via the PForDelta algorithm [30], we achieve a combined index and data column size of only 77–93% of the original column size.

2 Background

2.1 Indexing

Search and query processing operations on traditional database systems like Oracle, MySQL, and DB2 involve the use of indexing techniques that are usually variants of either bitmap indexes or $B-$trees. While these techniques are effective in speeding up query response times, they come at the cost of a heavy-weight index management scheme. Indexing with $B-$trees [9], which tends to be more suitable for transactional databases that require frequent updates, is observed to consume storage that is one-to-three times the size of the raw column data for high-cardinality attributes. Scientific data, which is typically read (or append) only, have been shown to be better served with bitmap-based indexing techniques [19, 23], providing faster response times with lower index storage overhead.

While there are numerous technologies that use variants of bitmap indexing, we primarily focus on FastBit [23], a state-of-the-art bitmap indexing scheme, that is used by a number of scientific applications for answering range queries. FastBit employs a Word-Aligned-Hybrid (WAH) compression scheme based on run-length encoding, which decreases the index storage requirement and allows FastBit to perform logical operations efficiently on the compressed index and compute partial results by scanning the index. While the required storage for WAH is larger than that for bitmap compression variants such as Byte-aligned Bitmap Compression [4], WAH has been shown to be far faster for query processing. For those records that cannot be evaluated with the index alone, FASTBIT resorts to performing a read of the raw data, in what is called *candidate checks*. Unfortunately, the bitmap index created is sensitive to the distribution and cardinality of the input data, taking anywhere from 30 to 300% of the raw column size. The space can partly be reduced through techniques such as *precision binning*, at the cost of disturbing the distribution of values along the bins.

Another type of index, popular for document indexing, is an inverted index [22, 29]. Traditionally, in document clustering systems, a single document is identified by terms, which typically correspond to some subset of written language. An inverted index, in this case, maps each term in the dictionary to the list of documents the term appears in, greatly speeding up queries constrained by term. In this work, an inverted mapping is used in a different context: we are mapping histogram bins to lists of records that fall within each bin.

2.2 Compression

Data compression methods within databases have been widely studied as an important component for lowering the storage footprint of data-stores [11, 14, 21]. For example, the column-oriented database C-Store [2] uses null compression (elimination of zeroes), dictionary encoding, and run-length encoding for effective data reduction of attributes organized contiguously, as opposed to the traditional row-store organization. While these methods have limited use on floating-point data due to high-entropy significand bits, our work does share similarity with the dictionary encoding method, in that we compress floating-point data through identifying unique values and assigning them reduced bitwise representations. However, we perform this on only the most significant few bytes of the floating-point data, as opposed to the full dataset as in C-Store, and discard the representation entirely when using the inverted index for our query processing methodology.

A compression methodology particularly important to our methods is based on inverted index compression techniques. Asides from general purpose compressors, specialized techniques, such as Simple9, Simple16, Relate10 and Carryover12 [3] provide high compression ratios while being computationally efficient for compressing and processing indexes on text collections. Furthermore, document reordering methods have been devised to increase locality, thereby achieving higher compression ratios [27]. For our method, a particularly important inverted index compression algorithm is the PFor family of compressors, which include PFor, PForDelta, and PDict [30], built specifically for fast compression and decompression speeds on modern CPU architectures. Each method uses reduced, fixed bit widths to encode values with a high degree of similarity (PFor and PForDelta) or commonly occuring values (PDict), encoding the remaining in full-precision as *exceptions*. Of these methods, PForDelta is the basis for our inverted index compression, as it first encodes differences between successive values before compressing. See Section 3.3.

As mentioned, many general-purpose and specialized compression methodologies fail to provide high compression ratios on floating-point data. Part of the reason for this is that floating-point scientific data is notoriously difficult to compress due to high entropy significands, of which floating-point data is primarily composed of (23 of 32 bits for single precision and 52 of 64 bits for double-precision). Much work has been done to build compressors for these kinds of data, mostly based on difference coding. Algorithms such as FPC [6] and fpzip [17] use *predictors* like the Lorenzo predictor [12], FCM [28] and DFCM [10] to compress. Given an input stream of floating-point values, the predictors use the previously seen values to predict the next value in the stream, and rather than attempt to compress the floating-point values themselves, the compression algorithm uses a measure of error between the predicted and actual value, typically as an XOR operation.

Our compression/indexing methodology is based on treating the most significant bytes of floating-point data differently than the least significant bytes. Isenburg *et al.* use the same underlying concept in a prediction-based compression utility, which partitions the sign, exponent, and significand bits of the prediction error, followed by compression of each component [13]. Unlike their method, our method must maintain the approximability of floating point datasets by treating the most significant bytes as a single component (sign, exponent, and the most significant significand bits), enabling efficient index generation and range query processing over the compressed data.

Another method that is based on processing data with respect to significant bytes is ISOBAR preconditioner [18]. Based on the observation that a significant byte-wise view of the data can yield patterns not picked up by existing compressors, ISOBAR first determines the *compressibility* of input data by looking at the frequency distribution on a significant byte level. Significant byte columns that appear to have a uniform frequency distribution (such as mantissa bytes in floating-point variables) are ignored in subsequent compression, leading to greatly increased compression speeds.

3 Method

3.1 System Overview

As mentioned, the goal of this paper is to facilitate query-driven analysis of large-scale scientific simulation data with storage-bound requirements. There are two stages where we focus our design to achieve this goal: first, while simulation data is being generated and is still in memory, or later as a post-processing step, we can process and reorganize a floating-point dataset to compress the data. Second, we can modify the new organization of data to optimize query processing on the preprocessed data. For this purpose, we introduce two components in the scientific knowledge discovery pipeline, the *lossless compressor/indexer* and the *query engine*. These correspond to two different use cases using the same underlying process – a compression-only use case and a query-processing use case.

3.2 Compression

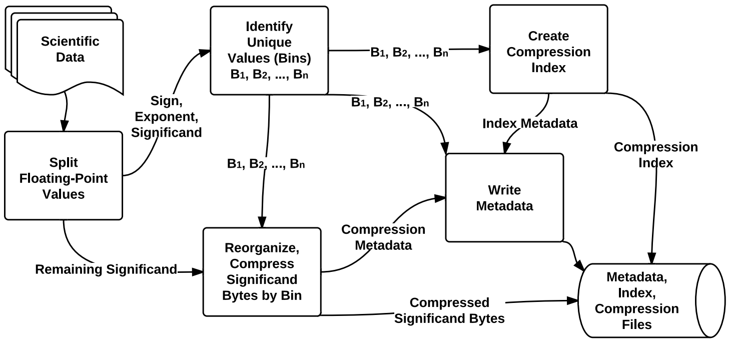
Scientific simulations use predominantly double-precision floating-point variables, so the remainder of the paper will focus on compression and query processing for these variables, though our method can be applied to floating point numbers of different precision. The underlying representation of these variables, the IEEE 754 floating-point standard [1], is a primary driver of our compression and querying methodology, so we briefly review it here. The standard encodes floating point values using three components: a *sign* bit, an *exponent* field, and a *significand* field. For example, 64-bit double-precision values use one sign bit, 11 exponent bits, and 52 significand bits. Given the sign bit s, the unsigned integral representation of the exponent field e, and each significand bit m_i (most to least significant), the resulting value encoded by a double-precision variable is:

$$\text{value} = (-1)^s \times 2^{e-1023} \times (1 + \sum_{i=1}^{52} (m_i 2^{-i})). \tag{1}$$

Note that, all other components being equal, a difference of one in the exponent fields of two double-precision numbers leads to a 2x difference in the represented values.

Our key observation for the compression process is that there is similarity among values in our target datasets with respect to orders of magnitude. For instance, in a simulation grid, adjacent grid values are unlikely to differ in orders of magnitude, except perhaps along simulation-specific phenomenon boundaries. Furthermore, the encoding naturally lends itself to accurate approximation given the exponent components. Hence, we base our compression and query processing methodology on the commonality in the sign and exponent field of double-precision datasets.

Figure 1 gives an overview of the compression process, developed under the assumption of similar exponent components and with the goal of being amenable to range query processing. For an N-element *partition* (a single block of configurable size from the dataset, to be compressed and indexed as a unit), we split the $8N$-byte double-precision column stream into two components: a kN-byte *high-order byte stream* consisting of the most significant k bytes of each value, and the remaining $(8-k)N$-byte *low-order byte stream* consisting of the remaining significant bytes. Using the observation of highly similar sign and exponent values, we identify the unique high-order bytes and discard redundant values. Let n be the number of unique high-order byte patterns. We define a *bin* to be a set of low-order bytes with equivalent high-order bytes, with bin edges



Scientific Data
Split Floating-Point Values
Sign, Exponent, Significand
Identify Unique Values (Bins) B1, B2, ..., Bn
B1, B2, ..., Bn
Create Compression Index
B1, B2, ..., Bn
Index Metadata
Compression Index
B1, B2, ..., Bn
Write Metadata
Compression Metadata
Remaining Significand
Reorganize, Compress Significand Bytes by Bin
Compressed Significand Bytes
Metadata, Index, Compression Files

Fig. 1. Various stages of the compression methodology, described in Section 3.2. The bitmap index is used for compression, while the inverted index is used in query processing.

$B_1, B_2, \ldots, B_n$ corresponding to the sorted unique patterns. Figure 2 shows the relationship between floating-point values, the high and low-order bytes, and their resulting bins. The low-order bytes are reorganized into their respective bins, and a record ID (RID) to bin mapping M is generated to maintain the original organization, using a bitmap with $\lceil \log(n) \rceil$ bits per identifier. A general-purpose compressor (such as bzip2) is then run on the bin mapping M, as well as (optionally) the low-order bytes.

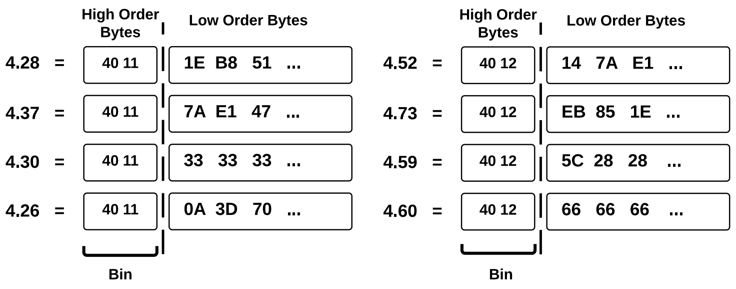

High Order Bytes
Low Order Bytes
4.28 = 40 11 1E B8 51 ...
4.37 = 40 11 7A E1 47 ...
4.30 = 40 11 33 33 33 ...
4.26 = 40 11 0A 3D 70 ...
Bin
High Order Bytes
Low Order Bytes
4.52 = 40 12 14 7A E1 ...
4.73 = 40 12 EB 85 1E ...
4.59 = 40 12 5C 28 28 ...
4.60 = 40 12 66 66 66 ...
Bin

Fig. 2. Mapping between floating-point numbers, high- and low-order bytes, and their respective bins

Three data structures are produced as the result of the compression process: (1) the compression metadata, defining the high-order byte values and file offsets of each bin, (2) the compressed RID-to-bin mapping M, and (3) the bin-organized low-order bytes.

The value of k should be chosen with two goals in mind: to cause the number of distinct high-order bytes to stabilize with an increasing stream size, and to maximize the redundancy of the patterns (for compression) while encoding the entirety of the sign and exponent components (for future query processing). For scientific floating point data, we have found $k = 2$ to be the most effective; it covers the sign bit, all exponent bits, and the first four significand bits of double-precision values (approximately two significant figures in base 10 scientific notation). This makes sense, as higher degrees of precision in scientific data tend toward high-entropy values. To verify our choice of k for this paper, Figure 3 shows the number of distinct high-order bytes recorded as a

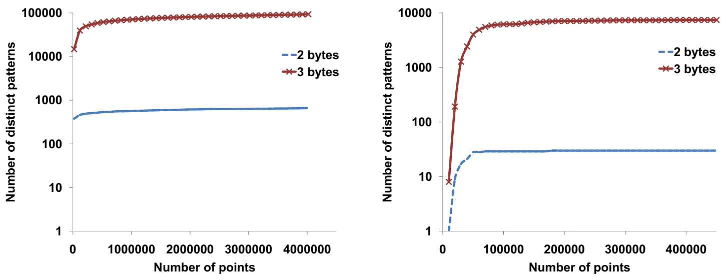


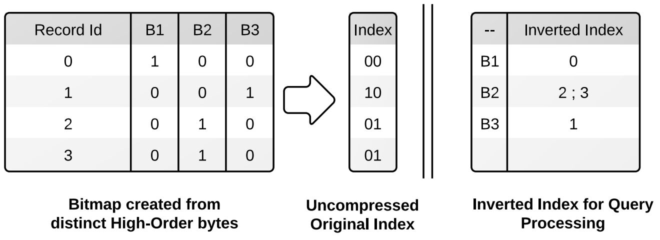
Fig. 3. Cumulative growth of the number of distinct higher order 2-byte and 3-byte pattern for increasing data size

data stream is processed. For both $k = 2$ and 3, a relatively small cardinality is seen relative to the number of points processed, with the distinct values quickly reaching a (near) maximum.

Recall that the metadata consists of unique high-order bytes as well as their respective file offsets to the low-order byte payload. Hence, the metadata size is directly proportional to the number of unique high-order bytes. As shown in Figure 3, for two of the scientific datasets, the size of metadata is less than 0.1% of the dataset for $k = 2$, due to the small number of distinct patterns. For $k = 3$, however, the number of distinct patterns increases by a factor of 100 due to the addition of the higher-entropy significand bits. This increases the metadata size similarly, while additionally increasing the size of the RID to bin mapping logarithmically. Given the trends in Figure 3, we expect random sampling to be sufficient to determine a good value of k for double-precision datasets.

3.3 Query Processing: Index Generation

The compression methodology presented in Section 3.2 is, as will be shown, effective at improving the compression ratio of many scientific datasets, but is not optimized for query processing. If a range query is performed using our compression index, the entire RID-to-bin mapping M would need to be traversed to map the binned data back to RIDs. Thus, we develop another method to optimize for range queries by using an inverted index, at the cost of additional storage. This inverted index, which we denote as M^{-1}, maps each bin to a list of RIDs sharing the same high-order bytes, creating a *bin-based value-to-RID* mapping. Figure 4 illustrates the index used in compression compared to the inverted index. This organization is advantageous for range query processing because we now access the RIDs by bin (the same access pattern as with the low-order bytes). It is initially disadvantageous, however, because of the increased space. This means, for a partition of N elements, approximately $N\log(N)$ bits are needed to store the index, with marginal additional space to store metadata such as the number of elements within each bin. Bounding the maximum partition size to 32GB of double-precision data ensures that each RID in the inverted index needs no more than four bytes, making the index size less than 50% of the raw column size, or lower for smaller partitions. As a simple example, a partition size of 2GB of double-precision data requires 28 bits for each RID, translating to an index size of 43.75% of the raw column size. This is assuming, of course, that the partition is completely filled.



Record Id	B1	B2	B3
0	1	0	0
1	0	0	1
2	0	1	0
3	0	1	0

Index
00
10
01
01

--	Inverted Index
B1	0
B2	2 ; 3
B3	1

Bitmap created from distinct High-Order bytes
Uncompressed Original Index
Inverted Index for Query Processing

Fig. 4. Building an inverted index for query processing from the index used in compression

Inverted Index Compression. While the index is relatively small compared to the column size, we make a few observations that allow us to further reduce index storage overhead. Our inverted index works on the bin-level by using linearized RIDs, with the resulting structure of an ordered list of RIDs. This presents the perfect opportunity to use a compressed inverted index, based on difference (delta) encoding. Specifically, we use the PForDelta algorithm. Given a desired bits-per-item parameter b, PForDelta stores a *base* value (the first RID in the sorted list), and differences between consecutive elements, using only b bits per-difference. For values that cannot be stored using b bits (namely, differences greater than $2^b - 1$), that element, called an *exception*, is stored at the end of the compression block, with a special marker put in it's place. Then, for decompression, a two-pass approach is taken: the first pass restores all deltas to their original state, and the second corrects, or "patches" the values encoded as exceptions, while computing the running sum and restoring values. The PForDelta default block size is used for compression (128 elements), for both cache efficiency and to allow the value of b to vary across blocks.

For PForDelta compression to be effective, the parameter b must be determined to optimize and balance compression ratio and speed. While we do not provide a detailed evaluation, we use the approach from the original PForDelta paper: select the b that results in the highest compression ratio by computing, for each compression block, the value of b that minimizes total size (compressed delta list + exception list). For the datasets evaluated in this paper, this heuristic represents a good tradeoff between compression ratio and speed, because a large number of exceptions hurts both metrics.

Low-order Byte Compression with ISOBAR. The need for fast (de)compression speeds in the indexing process requires us to revisit the idea of low-order byte compression – for the compression target of our method, the primary metric is instead storage reduction. Previous work [15] showed two trends using low-order byte compression along with indexing: that compression gains for the low-order bytes vary widely across datasets, and that compression speed was the limiting factor for indexing speed. We believe these trends arise because the underlying data being compressed is composed of entirely floating-point mantissa bytes, which tend to have a more uniform distribution.

Based on these observations, we use the ISOBAR preconditioner to cut down on compression costs, while still receiving the benefits of data reduction. Figure 5 shows the modified compression process. Each significant byte-column is analyzed for compressibility using frequency analysis. Each column deemed compressible by ISOBAR is then compressed, while the "incompressible" columns are kept as-is, saving on computation.

3.4 Query Processing: File Layout

The data used by the query processing engine is split into three components: a metadata file, an index file, and a compression file, each corresponding to its purpose described in the previous sections.

The metadata file layout is shown in Figure 6. The metadata file contains partition information, including file offsets for each partition and bin, the number and bounds (high-order bytes) of bins, and the number of values per bin per partition. The index

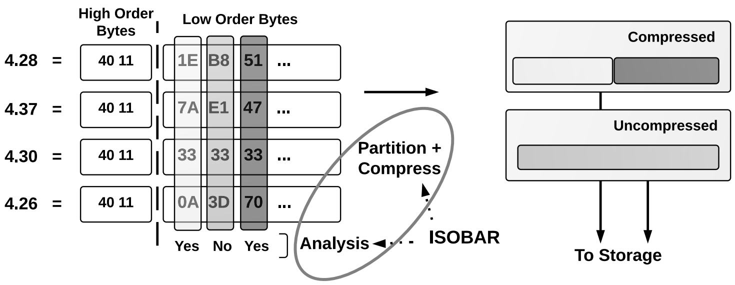


Fig. 5. ISOBAR analysis and compression, applied on a per-bin basis

```
<N number of partitions>
<Metadata offset for partition t> (0 ≤ t < N)
<Index offset, state flag for partition t> (0 ≤ t < N)
<Low order byte offset, state flag for partition t> (0 ≤ t < N)
(Repeat for 0 ≤ t < N)
<P number of elements in partition t>
<B number of bins>
<Number of elements in bin b> (0 ≤ b < B)
<Bin bound b> (0 ≤ b < B)
<Compression offset b> (0 ≤ b < B)
(End Repeat)
```

Fig. 6. Metadata file format

file and the compression file contain the RIDs and compressed low-order bytes, respectively. A single scan of the metadata file is necessary for query processing and is small enough to be held in memory to optimize future queries. In our experimentation, however, we do not consider this possibility.

3.5 Query Processing: Range Queries

The processing of range queries is based on two characteristics of our compression/indexing process: that the bins (low-order bytes and inverted index) are organized on disk in increasing order of high-order bytes, and that bin edges (the high-order bytes) provide a lower bound on the values of RIDs within each bin by treating the high-order bytes as a truncated double-precision value.

The query evaluation process is shown in Figure 7. Given a variable constraint $[v_1, v_2)$, the metadata file shown in Figure 6 is traversed to obtain the necessary high-order bytes and bin file-offsets. Using the high-order bytes as a lower-bound for values within a bin, the boundary bins B_x and B_y are obtained using a binary search. Then, a single, contiguous read is performed per partition in each of the index and low-order bytes files in order to fetch the data corresponding to the range of bins $B_x, B_{x+1}, \ldots, B_y$, taking advantage of the bin organization in file. The column data corresponding to the

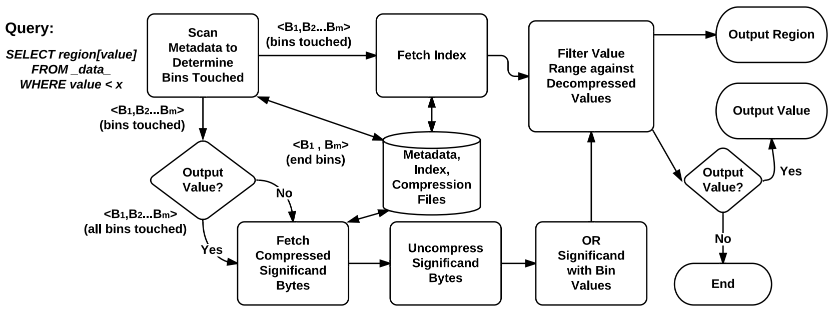

Query:
SELECT region[value] FROM _data_ WHERE value < x
Scan Metadata to Determine Bins Touched
<B1,B2...Bm> (bins touched)
Fetch Index
Filter Value Range against Decompressed Values
Output Region
Output Value
<B1,B2...Bm> (bins touched)
Output Value?
<B1 , Bm> (end bins)
Metadata, Index, Compression Files
No
<B1,B2...Bm> (all bins touched)
Yes
Fetch Compressed Significand Bytes
Uncompress Significand Bytes
OR Significand with Bin Values
Output Value?
Yes
No
End

Fig. 7. Query processing methodology, taking into account metadata, index, and compression data fetching and aggregating

low-order bytes are reconstructed and only the data in boundary bins are filtered against the query bounds.

In the case of queries requesting only RIDs, not all of the low-order bytes need to be fetched and reconstructed. Only the bins at each boundary need be read and checked against the query constraints, as all remaining bins are guaranteed to fit within the query bounds.

4 Results and Discussions

4.1 Experimental Setup

We performed our experiments on the Lens cluster at Oak Ridge National Laboratory, dedicated to high-end visualization and data analysis. Each node in the cluster is made up of four quad-core 2.3 GHz AMD Opteron processors and is equipped with 64GB of memory. All experiments were run with data located on the Lustre filesystem. For the indexing and query processing experiments, we compare against WAH encoding as implemented by FastBit, version 1.3.4. To avoid database-related overheads such as concurrency control, transaction support, etc. and provide a fair comparison between technologies, we wrote a minimal query driver for FastBit using only the necessary indexing and querying functions provided in the FastBit API. Also in the interest of fairness, we use the same partition size of 2GB for both our method and FastBit.

4.2 Datasets

To evaluate our compression, indexing, and query processing performance, we use a collection of double precision datasets from various sources. The majority of the datasets (*msg*, *num*, and *obs*) are publicly available and discussed by Burtscher and Ratanaworabhan [7]. We additionally use timeslice data for numerous variables generated by the GTS [20], FLASH [5], S3D [8], and XGC-1 [16] simulations.

In particular, we used the following two scientific simulation datasets to evaluate our query performance in terms of value-centric queries and region-centric queries: 1) GTS [20], a particle-based simulation for studying plasma microturbulence in the core of magnetically confined fusion plasmas of toroidal devices, and 2) S3D [8], a first-principles-based direct numerical simulation (DNS) of reacting flows that aids the modeling and design of combustion devices.

4.3 Query Processing

Index Generation

We evaluate the performance of our index generation methodology with respect to both computational and storage efficiency. We utilize inverted index compression, but not low-order byte compression, when comparing our method to FASTBIT's, as this represents the best tradeoff between indexing speed and storage footprint. Table 1 shows the results obtained from these experiments.

As shown in Table 1, our combined index and data encoding outperforms numerous FASTBIT configurations with respect to both speed and storage. Even with index compression, ALACRITY is shown to encode the data at an order of magnitude higher rate

Table 1. Query index generation throughput and storage footprint. A_I: ALACRITY with inverted index compression. F_D: FASTBIT with default configuration (10^5 bins). $F_{2,3}$: FASTBIT with bin boundaries at two/three significant digits.

Dataset	**Index Gen. (MB/s)**				**Storage (data+index) Requirement (%)**			
	In-situ		**Post-proc.**					
	A_I	F_2	A_I	F_2	A_I	F_2	F_3	F_D
msg_bt	**120**	9	**67**	8	**87.3**	152.0	178.1	192.6
msg_lu	**125**	10	**65**	9	**87.2**	162.6	197.9	201.6
msg_sp	**136**	10	**73**	9	**83.0**	126.2	157.0	197.7
msg_sppm	**137**	12	**73**	11	**81.3**	114.7	116.8	125.3
msg_sweep3d	**140**	8	**63**	7	**85.0**	148.4	187.5	200.9
num_brain	**138**	9	**65**	8	**87.6**	164.3	191.5	202.3
num_comet	**107**	7	**53**	7	**92.9**	181.4	193.1	196.1
num_control	**109**	6	**58**	6	**93.1**	154.8	199.6	200.9
num_plasma	**130**	6	**38**	6	**86.6**	157.3	189.3	197.6
obs_error	**138**	11	**51**	10	**88.5**	149.9	167.6	176.9
obs_info	**132**	11	**28**	10	**85.0**	138.1	181.3	219.3
obs_spitzer	**137**	12	**69**	11	**87.0**	146.4	177.2	198.3
obs_temp	**121**	10	**40**	9	**91.9**	187.0	200.1	210.0
gts_phi_l	**111**	7	**42**	6	**92.9**	181.5	199.4	208.8
gts_phi_nl	**112**	7	**42**	6	**92.9**	183.6	199.7	208.9
gts_chkp_zeon	**110**	7	**28**	7	**91.3**	176.3	198.9	220.4
gts_chkp_zion	**114**	7	**28**	6	**89.9**	166.1	194.6	220.0
gts_potential	**112**	6	**71**	6	**92.5**	184.0	197.9	199.8
xgc_iphase	**105**	10	**68**	9	**90.0**	168.3	172.3	176.9
s3d_temp	**144**	14	**71**	13	**80.4**	117.2	135.4	202.0
s3d_vvel	**123**	11	**64**	10	**90.1**	171.7	195.0	202.1
flash_velx	**101**	9	**80**	8	**82.4**	123.8	157.2	195.7
flash_vely	**107**	9	**83**	9	**79.4**	112.3	137.3	193.1
flash_gamc	**110**	16	**83**	14	**77.5**	100.4	102.1	198.1

than the fastest configuration of FastBit we tested, though the gap narrows somewhat when including read/write measurements: computing the compressed bitmap indexes appears to be FastBit's rate-limiting factor.

To show a full picture of ALACRITY's performance characteristics with respect to data and index encoding, Table 2 shows indexing results over the same datasets, this time showing the effect of different configurations of ALACRITY. First, the use of index compression is a good way of reducing the overall storage footprint while not incurring high computational costs. Second, the efficacy of low-order byte compression shows widely varying storage results based on the particular underlying dataset. This is a result of the mantissa bytes being highly entropic, though ISOBAR improves the encoding speed when not much data is selected for compression. Third, the choice of k is highly important in the context of indexing performance. As increasing values of k tend to increase the number of unique patterns exponentially (see Figure 3), the resulting

Table 2. Query index generation throughput and storage footprint among multiple ALACRITY configurations. A_{IB}: ALACRITY with inverted index and low-order byte compression. A_B: ALACRITY with low-order byte compression only. A_I: ALACRITY with inverted index compression only. A_{I3}: ALACRITY with inverted index compression and $k = 3$. A: ALACRITY without additional compression.

Dataset	**Index Gen. (MB/s)**										**Storage (data+index) Requirement (%)**				
	In Situ					**Post-proc.**									
	A	A_I	A_{I3}	A_B	A_{IB}	A	A_I	A_{I3}	A_B	A_{IB}	A	A_I	A_{I3}	A_B	A_{IB}
msg_bt	**207**	120	44	28	25	**85**	67	33	23	21	125.0	87.3	97.7	118.7	**81.0**
msg_lu	**217**	125	42	26	24	**82**	65	30	22	20	125.0	87.2	99.6	124.4	**86.6**
msg_sp	**240**	136	57	33	30	**86**	73	42	27	25	125.0	83.0	91.5	120.7	**78.7**
msg_sppm	**224**	137	82	66	54	**86**	73	54	48	43	125.0	81.3	75.1	60.6	**16.8**
msg_sweep3d	**233**	140	56	32	29	**76**	63	38	25	23	125.0	85.0	92.5	105.1	**65.1**
num_brain	**243**	138	58	22	21	**77**	65	39	19	18	125.0	87.6	92.0	124.6	**87.2**
num_comet	**167**	107	26	32	29	**68**	53	20	24	23	125.0	92.9	114.2	114.0	**81.8**
num_control	**179**	109	33	27	24	**71**	58	25	22	20	125.0	93.1	117.2	124.1	**92.2**
num_plasma	**213**	130	66	107	79	**45**	38	30	38	33	125.0	86.6	86.9	50.9	**12.5**
obs_error	**243**	138	62	42	37	**56**	51	37	28	25	125.0	88.5	91.3	88.1	**51.6**
obs_info	**226**	132	37	52	45	**38**	28	17	21	18	125.0	85.0	114.5	77.8	**37.8**
obs_spitzer	**251**	137	59	30	27	**85**	69	41	24	23	125.0	87.0	87.3	94.8	**56.8**
obs_temp	**206**	121	35	25	23	**55**	40	22	17	16	125.0	**91.9**	109.3	125.1	91.9
gts_phi_l	**184**	111	33	36	32	**81**	42	19	23	21	125.0	**92.9**	122.1	125.1	93.0
gts_phi_nl	**186**	112	34	41	36	**88**	42	22	25	24	125.0	**92.9**	117.3	125.0	92.9
gts_chkp_zeon	**181**	110	19	28	26	**83**	27	11	16	15	125.0	**91.3**	136.3	125.2	91.4
gts_chkp_zion	**191**	114	20	28	25	**85**	28	12	17	15	125.0	**89.9**	133.3	125.1	90.0
gts_potential	**181**	112	33	48	40	**87**	71	28	37	33	125.0	92.5	95.5	124.9	**92.4**
xgc_iphase	**150**	105	28	29	26	**80**	68	25	25	23	125.0	90.0	88.0	105.4	**70.4**
s3d_temp	**267**	144	100	48	42	**106**	70	58	35	32	125.0	80.4	76.9	118.7	**74.2**
s3d_vvel	**216**	123	43	23	21	**97**	64	32	19	18	125.0	**90.1**	97.4	125.0	90.2
flash_velx	**267**	142	78	23	21	**101**	80	52	20	19	125.0	82.4	90.1	125.0	**82.4**
flash_vely	**246**	144	80	23	21	**107**	83	54	20	19	125.0	79.4	89.2	125.0	**79.4**
flash_gamc	**270**	148	142	96	74	**110**	83	83	60	54	125.0	77.5	68.7	115.1	**67.6**

effect on indexing performance is negative in both generation time and storage overhead. However, it is useful in query processing, as it minimizes the cost of processing misaligned bins.

In our previous work, we utilized *zlib* for the low-order byte compression. A detailed comparison between the use of ISOBAR and zlib can be seen in Table 3. Note that the underlying compressor used by ISOBAR in this work is actually zlib; ISOBAR is technically a compression preconditioner. The use of ISOBAR on the same datasets produced a speed increase while leaving the storage footprint virtually unchanged. Specifically, we see a median 22% (mean 40%) increase in encoding throughput, with a corresponding median 0.01% (mean −0.31%) increase in storage.

Table 3. ALACRITY indexing performance, using ISOBAR and zlib as the underlying compressors

Dataset	**Index Gen. (MB/s)**		**Storage (data+index)**	
	In Situ		**Requirement (%)**	
	w/ISOBAR	w/zlib	w/ISOBAR	w/zlib
msg_bt	**28**	21	**118.7**	119.4
msg_lu	**26**	21	**124.4**	124.4
msg_sp	**33**	20	**120.7**	124.0
msg_sppm	**66**	37	60.6	**59.6**
msg_sweep3d	**32**	22	105.1	**96.6**
num_brain	**22**	20	124.6	**124.5**
num_comet	**32**	17	**114.0**	116.2
num_control	**27**	21	124.1	**124.1**
num_plasma	**107**	62	**50.9**	51.4
obs_error	**42**	30	**88.1**	94.9
obs_info	**52**	37	77.8	**75.1**
obs_spitzer	**30**	20	94.8	**94.4**
obs_temp	**25**	21	125.1	**125.0**
gts_phi_l	**36**	21	125.1	**125.0**
gts_phi_nl	**41**	21	125.0	**125.0**
gts_chkp_zeon	**28**	21	125.2	**125.1**
gts_chkp_zion	**28**	21	125.1	**125.1**
gts_potential	**48**	20	**124.9**	125.0
xgc_iphase	**29**	22	105.4	**105.3**
s3d_temp	**48**	19	**118.7**	123.3
s3d_vvel	**23**	20	125.0	**125.0**
flash_velx	**80**	21	**125.0**	125.0
flash_vely	**83**	21	**125.0**	125.0
flash_gamc	**83**	17	**115.1**	121.4

End-to-End Query Performance Evaluation

For an end-to-end performance comparison, we perform queries under a number of scenarios, using the GTS potential (gts_potential) and S3D temperature (s3d_temp) variables. We look at two types of range queries: those that return record IDs given constraints on variables (which we refer to as "region-centric" queries, as they are used to retrieve "regions of interest"), and those that additionally output the values of the variables (which we will refer to as "value-centric" queries). We compare ALACRITY on each of these query types against FastBit, which is optimized for range queries, especially those of the "region-centric" type.

For both types of queries, we use ALACRITY with index compression, and ALACRITY without it. For region-centric queries, we use $k = 3$ and FastBit precision-3 binning (e.g., bin boundaries use three significant figures). This is so we can avoid performing costly candidate checks in both FastBit and ALACRITY and evaluate query processing with only the index, corresponding to fully "aligned" queries. For value-centric queries, we use $k = 2$ and FastBit precision-2 binning. This is done because value-centric queries are dominated by data retrieval, and the lower-precision indexes incur lower time to process, at the cost of having higher false positives. As the majority of the data read in tends to satisfy query constraints due to binning, the cost of pruning false positives is outweighed by the benefit of a lighter index.

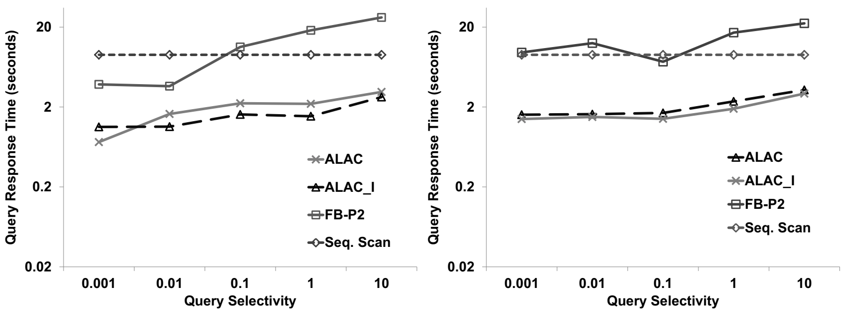


Fig. 8. Comparison of query response times for varying query selectivity, between ALACRITY (ALAC), ALACRITY with compressed inverted indexes (ALAC_I), FastBit (FB-P2) sequential scan. The left plot is for S3D temperature, while the right plot is for GTS potential.

Value-centric Queries. Figure 8 shows value-based query response time using our method, compared to FastBit's precision-based indexing (the fastest configuration we tested), with varying query selectivity. By query selectivity, we refer to the percentage of the column data returned by a query. For the GTS potential column, we provide a speedup in the range of 3.2 to 11.9. For the S3D temperature column, a speedup of 5.2 to 9.0 is observed. Due to the clustering of the data, a very small number of I/O seek operations are needed by our method relative to FastBit. Furthermore, the amount of data read by our method is much lower than that by FastBit, as shown in Table 1. The reason that sequential scan performs better than FastBit in this context is that, in parallel file systems such as Lustre, seeks are a very high-latency operation. For value-centric queries, FastBit incurs a seek per item, whereas sequential scan reads all data in a single, large read, and so for less selective queries, the seek costs outweigh the read costs.

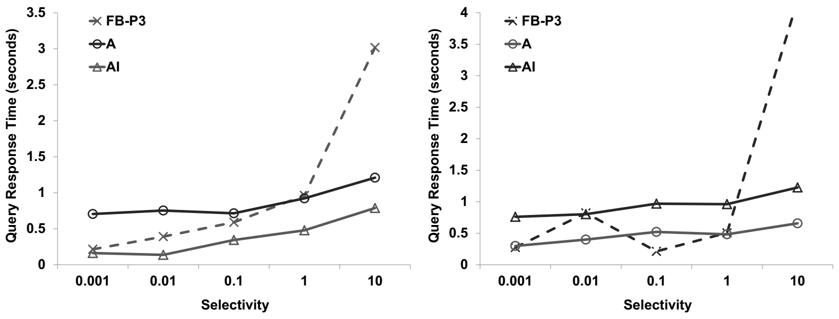


Fig. 9. Comparison of response return by FASTBIT (FB-P3) against ALACRITY, with and without inverted index compression (ALAC_I and ALAC, respsectively), for region-centric queries with varying number of query hits. The left and right plot show results for S3D temperature and GTS potential, respectively.

Region-centric Queries. Figure 9 shows region query response time with varying number of hits (records returned) for our method compared to FASTBIT with precision and default binning. As mentioned, the queries were chosen so that no candidate checks are needed. The precision-based boundaries are the basis for our method and is base two, while for FASTBIT it is a special configuration using base ten. Overall, it is seen that query performance time is low in nearly all cases on account of only needing to process the index, but different configurations of ALACRITY are able to meet or slightly outperform the FASTBIT alternative, likely due to the lower index size, though decompression overhead is a concern, as shown in the right plot of Figure 9. For the GTS potential variable, the compressed index size for $k = 3$ was less space-efficient than that for $k = 2$, making the read+decompress overhead larger than just reading the raw inverted index.

4.4 Performance Analysis

Figure 10 shows the breakup of overall query processing time into I/O and compute components, corresponding to index/bin loading and processing, respectively. The dataset tested on is S3D using the velocity variable. I/O is the dominant cost of query processing, while the application of the query constraints and data transformations is a low, though non-negligible, component. We believe multithreading or asynchronous I/O would be able to hide most of the compute costs by interleaving it with the more costly I/O operations.

4.5 Compression

To analyze the performance of our lossless data compression scheme, we compare the compression ratios obtained with our method (without the inverted index) to those obtained by general-purpose lossless compression utilities, as well as more recent floating-point compressors. Out of the datasets tested, our method performed better than all of

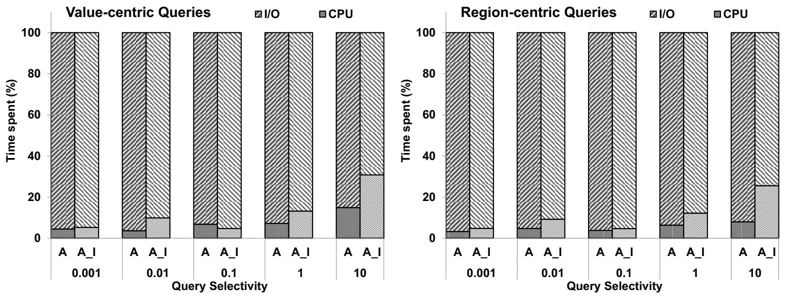


Fig. 10. Comparison of computation and I/O time distribution for ALACRITY for different query types of varying selectivity, on the S3D temperature variable. A: ALACRITY without bin or inverted index compression. A_I: ALACRITY with inverted index compression.

Table 4. Compression ratio and ALACRITY storage components. A_B: ALACRITY with bin compression (using bzip2).

Dataset	**Compression Ratio**					**Storage Requirement (%)**		
	gzip	**fpzip**	**bzip2**	**FPC**	**A_B**	**Data**	**Index**	**Metadata**
msg_bt	1.12	1.20	1.09	1.29	**1.40**	69.35	1.86	≈0.00
msg_lu	1.05	1.13	1.01	1.17	**1.30**	74.42	1.97	0.01
msg_sp	1.10	1.11	1.06	1.26	**1.33**	73.98	1.11	≈0.00
msg_sppm	7.41	3.25	7.09	5.30	**8.87**	9.58	1.66	0.02
msg_sweep3d	1.09	1.33	1.32	**3.09**	2.11	46.60	0.67	0.02
num_brain	1.06	1.25	1.06	1.16	**1.28**	74.50	3.39	≈0.00
num_comet	1.16	1.27	1.17	1.16	**1.34**	66.16	8.16	0.03
num_control	1.05	1.12	1.03	1.05	**1.15**	74.02	12.22	0.02
num_plasma	1.77	1.06	6.17	15.05	**75.72**	0.70	0.60	0.03
obs_error	1.44	1.37	1.36	**3.60**	2.59	34.07	4.51	≈0.00
obs_info	1.14	1.06	1.22	2.27	**3.52**	24.97	3.36	0.04
obs_spitzer	1.23	1.07	1.78	1.03	**1.90**	44.36	8.05	≈0.00
obs_temp	1.03	1.09	1.03	1.02	**1.13**	75.00	12.70	0.03
gts_phi_l	1.04	1.18	1.02	1.07	**1.19**	75.00	8.56	0.03
gts_phi_nl	1.04	1.17	1.01	1.07	**1.19**	75.00	9.20	0.03
gts_chkp_zeon	1.04	1.09	1.02	1.01	**1.17**	75.00	10.04	0.10
gts_chkp_zion	1.04	1.10	1.02	1.02	**1.18**	75.00	9.60	0.11
gts_potential	1.04	1.15	1.01	1.06	**1.18**	75.00	9.60	≈0.00
xgc_iphase	1.36	1.53	1.37	1.36	**1.58**	55.33	7.56	≈0.00
s3d_temp	1.18	**1.46**	1.15	1.34	1.35	73.38	0.77	≈0.00
s3d_vvel	1.04	1.24	1.02	1.15	**1.27**	75.00	3.74	≈0.00
flash_velx	1.11	**1.34**	1.08	1.26	1.32	75.00	0.81	≈0.00
flash_vely	1.13	**1.43**	1.09	1.29	1.32	75.00	0.80	≈0.00
flash_gamc	1.28	**1.62**	1.28	1.53	1.40	71.37	0.06	≈0.00

the other compressors tested (gzip, fpzip [17], bzip2, and FPC [7]) on 18 of 24. FPC gave superior performance compared to our method on two of the 27 datasets, while fpzip gave better performance on the remaining four. Overall, our method was consistent in yielding comparable or better compression ratios than the other compressors, providing evidence of strong compression ratios in other application datasets.

To justify our superior performance on most of the datasets, we argue that the bin-based compression of the data generally allows a much greater exploitation of existing compression algorithms than the normal distribution of scientific data that was passed to the other compressors. The reorganization of the data allowed bzip2 to be utilized as best as possible, causing the data to be reduced significantly because of the splitting of the low-entropy and high-entropy sections of the data. As evidenced by the small compressed index and metadata sizes, the reorganization is a low-overhead operation with respect to storage. We attribute the better performance of FPC and fpzip on some of the datasets to the encoding of data dependency which the FCM [28], DFCM [10], and Lorenzo [12] predictors used by FPC and fpzip were able to capture in their predictions.

5 Conclusion

As the size of scientific datasets in various disciplines continues to grow, new methods to store and analyze the datasets must be developed, as I/O capabilities are not growing as quickly, and new technologies (such as SSDs) are not currently able to achieve the storage density and cost-efficiency of traditional mechanical disk drives. Successful methods of mitigating this growing gap must involve data reduction in all stages of the knowledge discovery pipeline, including storage of raw data as well as analytics metadata. We believe our work in this paper in compression, indexing, and query processing of scientific data represents a step in the right direction, allowing both efficient lossless compression of floating-point data for accuracy-sensitive applications as well as efficient query processing on variable constraints, all with less space and I/O requirements than other database technologies.

Acknowledgements. We would like to acknowledge the use of resources at ORNL's leadership class computing facility, OLCF. Also, we appreciate the use of the datasets available from the Flash Center for Computational Science. This work was supported in part by the U.S. Department of Energy, Office of Science and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs). Oak Ridge National Laboratory is managed by UT- Battelle for the LLC U.S. D.O.E. under contract no. DEAC05- 00OR22725.

References

1. IEEE standard for floating-point arithmetic. IEEE Standard 754-2008 (2008)
2. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 671–682. ACM, New York (2006)

3. Anh, V.N., Moffat, A.: Index compression using fixed binary codewords. In: Proceedings of the 15th Australasian Database Conference, ADC 2004, vol. 27, pp. 61–67. Australian Computer Society, Inc., Darlinghurst (2004)
4. Antoshenkov, G.: Byte-aligned bitmap compression. In: Data Compression Conference, p. 476 (1995)
5. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series 131, 273–334 (2000)
6. Burtscher, M., Ratanaworabhan, P.: High throughput compression of double-precision floating-point data. In: IEEE Data Compression Conference, pp. 293–302 (2007)
7. Burtscher, M., Ratanaworabhan, P.: FPC: A high-speed compressor for double-precision floating-point data. IEEE Transactions on Computers 58, 18–31 (2009)
8. Chen, J.H., Choudhary, A., Supinski, B., DeVries, M., Hawkes, E.R., Klasky, S., Liao, W., Ma, K., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yoo, C.: Terascale direct numerical simulations of turbulent combustion using S3D. Comp. Sci. and Discovery 2(1)
9. Comer, D.: The ubiquitous B-Tree. ACM Comput. Surv. 11, 121–137 (1979)
10. Goeman, B., Vandierendonck, H., Bosschere, K.D.: Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In: Seventh International Symposium on High Performance Computer Architecture, pp. 207–216 (2001)
11. Graefe, G., Shapiro, L.: Data compression and database performance. In: Proceedings of the 1991 Symposium on Applied Computing, pp. 22–27 (April 1991)
12. Ibarria, L., Lindstrom, P., Rossignac, J., Szymczak, A.: Out-of-core compression and decompression of large *n*-dimensional scalar fields. Computer Graphics Forum 22, 343–348 (2003)
13. Isenburg, M., Lindstrom, P., Snoeyink, J.: Lossless compression of predicted floating-point geometry. Computer-Aided Design 37(8), 869–877 (2005); CAD 2004 Special Issue: Modelling and Geometry Representations for CAD
14. Iyer, B.R., Wilhite, D.: Data compression support in databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB 1994, pp. 695–704. Morgan Kaufmann Publishers Inc., San Francisco (1994)
15. Jenkins, J., et al.: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part II. LNCS, vol. 7447, pp. 16–30. Springer, Heidelberg (2012)
16. Ku, S., Chang, C., Diamond, P.: Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic Tokamak geometry. Nuclear Fusion 49(11), 115021 (2009)
17. Lindstrom, P., Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Transactions on Visualization and Computer Graphics 12, 1245–1250 (2006)
18. Schendel, E.R., Jin, Y., Shah, N., Chen, J., Chang, C., Ku, S.-H., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: ISOBAR preconditioner for effective and high-throughput lossless data compression. In: Proceedings of the 28th International Conference on Data Engineering, ICDE 2012. IEEE (2012)
19. Sinha, R.R., Winslett, M.: Multi-resolution bitmap indexes for scientific data. ACM Trans. Database Syst. 32 (2007)
20. Wang, W.X., Lin, Z., Tang, W.M., Lee, W.W., Ethier, S., Lewandowski, J.L.V., Rewoldt, G., Hahm, T.S., Manickam, J.: Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. Physics of Plasmas 13(9), 092505 (2006)
21. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. SIGMOD Rec. 29(3), 55–67 (2000)

22. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann (1999)
23. Wu, K.: Fastbit: an efficient indexing technology for accelerating data-intensive science. Journal of Physics: Conference Series 16, 556 (2005)
24. Wu, K., Ahern, S., Bethel, E.W., Chen, J., Childs, H., Cormier-Michel, E., Geddes, C., Gu, J., Hagen, H., Hamann, B., Koegler, W., Lauret, J., Meredith, J., Messmer, P., Otoo, E., Perevoztchikov, V., Poskanzer, A., Prabhat, Rubel, O., Shoshani, A., Sim, A., Stockinger, K., Weber, G., Zhang, W.-M.: FastBit: interactively searching massive data. Journal of Physics: Conference Series 180(1), 012053 (2009)
25. Wu, K., Otoo, E., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proc. of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 24–35 (2004)
26. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. 31, 1–38 (2006)
27. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th International Conference on World Wide Web, WWW 2009, pp. 401–410. ACM, New York (2009)
28. Yiannakis, S., Smith, J.E.: The predictability of data values. In: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 30, pp. 248–258. IEEE Computer Society, Washington, DC (1997)
29. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2) (July 2006)
30. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, pp. 59–71. IEEE Computer Society, Washington, DC (2006)