EUREKA: Enabling Fine-Grained Access and Range Queries on Compressed Scientific Data via Data-Index Co-Compression

Ning Yan
Georgia State University
Atlanta, GA, USA
nyan1@student.gsu.edu

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi1@anl.gov

Kai Zhao
Florida State University
Tallahassee, FL, USA
kai.zhao@fsu.edu

Lipeng Wan
Georgia State University
Atlanta, GA, USA
lwan@gsu.edu

ABSTRACT

Handling large-scale scientific data in high-performance computing (HPC) environments poses significant challenges, including excessive I/O, high storage costs, and slow query performance. Traditional approaches often require full data decompression and scans, making them impractical for real-time or interactive analysis. To address these limitations, we introduce EUREKA, a unified data-index co-compression framework that enables fine-grained access and efficient range queries on compressed scientific datasets.

EUREKA integrates spatial domain decomposition with block-wise error-bounded lossy compression to support selective decompression. It constructs a hierarchical AVL-tree index during compression to capture block-level value ranges, enabling fast pruning during query execution. To reduce metadata overhead, the index itself is also compressed while ensuring recall-preserving results. Experiments on six diverse HPC simulation datasets show that EUREKA achieves up to 25× data compression and over 300× index compression, surpassing state-of-the-art compressors such as SZ3 and ZFP in rate-distortion performance. Additionally, EUREKA delivers over 30× speedup for low-selectivity range queries, making it a scalable and efficient solution for modern scientific data analysis.

PVLDB Reference Format:
Ning Yan, Sheng Di, Kai Zhao, and Lipeng Wan. EUREKA: Enabling Fine-Grained Access and Range Queries on Compressed Scientific Data via Data-Index Co-Compression. PVLDB, 19(4): 631 - 643, 2025.
doi:10.14778/3785297.3785305

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ningyan11/Eureka.

1 INTRODUCTION

The rapid advancement of scientific experiments and simulations has led to an unprecedented surge in data generation [21]. For example, the International Thermonuclear Experimental Reactor (ITER), a global nuclear fusion initiative [9], is expected to produce about 2 petabytes of data per day once fully operational. Its diagnostic systems, including high-resolution imaging, magnetic sensors, and neutron detectors, will generate continuous streams of plasma behavior data, requiring sophisticated storage and retrieval solutions. Similarly, the Energy Exascale Earth System Model (E3SM) project [29], sponsored by the U.S. Department of Energy (DOE), can generate multi-petabyte outputs from climate simulations on DOE supercomputers. Analyzing such massive datasets is critical for advancing scientific discovery across domains such as climate science, fusion energy, and astrophysics.

Although scientific applications also produce diverse data types such as time series and trajectories, in this work we focus on array-based scientific data: large-scale, multi-dimensional arrays generated by numerical simulations (e.g., climate modeling, fluid dynamics, astrophysics, molecular dynamics) and high-resolution instruments (e.g., telescopes, particle detectors, microscopes). These datasets are dominated by dense floating-point fields encoding continuous physical quantities (e.g., temperature, velocity, density) and exhibit strong spatial and temporal correlations. Unlike transactional or social data, which is often discrete, unstructured, and managed by relational or document-oriented databases, or time series and trajectory data, which are lower-dimensional and handled with sequential or graph-oriented indexes, array-based scientific data demands fine-grained subarray access and value-range queries under strict accuracy requirements.

Given these characteristics, managing massive scientific datasets places extraordinary pressure on storage and I/O systems, making compression essential. Traditional lossless methods, while preserving exact fidelity, rarely achieve the high compression ratios required for practical use in large-scale scientific applications because floating-point data exhibits high entropy and minimal redundancy [20]. To overcome this limitation, error-bounded lossy compression has emerged as the preferred solution, offering much higher ratios while keeping errors within scientifically acceptable bounds. By allowing controlled approximations, these techniques trade a small amount of precision for substantial reductions in data size [27]. State-of-the-art compressors such as SZ [24, 32] and ZFP [12] routinely achieve 10:1 or higher ratios, cutting storage demands and accelerating data movement across computing nodes.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785305

They leverage domain knowledge to discard perceptually or computationally insignificant variations while preserving the essential structure and statistical properties of the scientific data.

While lossy compression significantly reduces storage footprints, it introduces a major challenge: efficient data access and analysis. In practice, scientists rarely need to analyze the entire dataset at once. Instead, they commonly perform a spatial or temporal decomposition, focusing on specific regions or time intervals that are most relevant to the investigation. For example, in fusion energy research, plasma studies may require extracting subsets of temperature and density over short time windows [15], while climate simulations often target regional events such as Arctic surface temperature anomalies or seasonal extremes [17]. Such fine-grained analyses demand selective retrieval of small subsets without decompressing full petabyte-scale outputs. However, existing lossy compressors lack this flexibility; they require full decompression even when only a small portion of the data is needed, leading to high I/O overhead, excessive memory use, and wasted computational resources, especially in large-scale scientific workflows.

In addition to enabling subset access, another critical challenge is supporting range queries on compressed data. Many scientific analyses require value-based filtering, such as retrieving all temperature readings within a range in a climate dataset or detecting high-energy plasma fluctuations in ITER diagnostics data. Without an efficient indexing mechanism, these queries typically involve decompressing the entire dataset followed by a full scan, resulting in significant computational overhead and latency. A natural solution is to incorporate indexing structures that enable fast, selective queries. However, indexing introduces a trade-off: while it improves query performance, it also consumes additional storage and reduces the overall compression ratio. This trade-off is especially pronounced when indexes are built at fine granularity to support precise queries, since higher resolution leads to larger index sizes and greater storage overhead.

In this paper, we aim to significantly reduce the storage footprint of both scientific data and their associated indexes while enabling fine-grained access and efficient range queries on compressed datasets. Unlike prior work that optimizes either raw data compression or index structures in isolation, we introduce a unified co-compression framework with structure-aware schemes tailored to scientific arrays and hierarchical indexes. This design addresses the limitations of traditional compression methods that require full decompression even for small queries, while also minimizing index overhead. As a result, our framework achieves compact storage, guarantees error-bounded reconstruction, and supports recall-preserving range queries. The key contributions are summarized as follows:

- We introduce Eureka, a novel data–index co-compression framework that applies error-bounded lossy compression to both raw scientific data and their indexing structures in a unified manner.
- We incorporate a domain decomposition strategy into the lossy compression process by first dividing large, multi-dimensional scientific datasets into uniform blocks and then compressing each block independently. This enables selective decompression and significantly improves access efficiency.
- We design a hierarchical indexing structure using an AVL tree, where each node represents a data block's value range. This structure accelerates range queries by pruning non-relevant blocks early in the search process. To minimize construction overhead, we exploit the traversal performed during block compression to compute minimum and maximum values for each block at no extra cost.
- We apply lossy compression to the index itself to reduce storage overhead, ensuring recall-preserving range queries in which all blocks overlapping the query range are retrieved. While false positives may occur due to approximation errors, they can be filtered during a secondary scan without affecting correctness.
- We evaluate Eureka on datasets from diverse scientific applications. The results demonstrate that Eureka achieves up to 25× compression on raw data and over 300× compression on index structures. Furthermore, it outperforms state-of-the-art compressors such as SZ3 [24] and ZFP [12] in rate–distortion tradeoffs and achieves over 30× speedup for low-selectivity range queries through its index-assisted query engine.

2 PROBLEM STATEMENT

We model the scientific dataset as a multi-dimensional array $D = \{d_1, d_2, \ldots, d_N\}$, where each $d_i \in \mathbb{R}$ is a floating-point value, and N is the total number of data points. The dataset is partitioned into B disjoint blocks, $D = \overset{B}{_{j=1}} D_j$, where each block D_j contains a contiguous subset of values aligned to spatial coordinates.

The data compression problem is to construct a compressed representation $C = fD, \varepsilon$, where f is an error-bounded lossy compressor parameterized by a user-defined error bound $\varepsilon > 0$. The decompressed data $\hat{D} = gC$ must satisfy the constraint $\max_{i=1,\ldots,N} |d_i - \hat{d}_i| \leq \varepsilon$. During block-wise compression, we also record the minimum and maximum values of each block, denoted as $l_j = \min D_j, h_j = \max D_j$. Collectively, these value ranges define the raw index $I = \{l_j, h_j | j = 1, \ldots, B\}$. We then organize I into an AVL-tree index, where each node corresponds to a block and stores its range l_j, h_j, along with an augmented attribute max_high for efficient pruning during queries.

The index compression problem is to reduce the storage of I while ensuring recall-preserving range queries. Formally, given a query interval α, β, the index must return a candidate set of blocks $Q_I \alpha, \beta = \{j | l_j, h_j \, \alpha, \beta \neq \emptyset\}$, such that all true matches are included: $\{j | \exists d \subset D_j, \alpha \leq d \leq \beta\} \subseteq Q_I \alpha, \beta$. False positives are permitted, but they are resolved during block decompression and value-level filtering.

Thus, the overall research problem addressed by this paper is to design a co-compression framework that minimizes storage for both C and the compressed index I, while guaranteeing error-bounded reconstruction and recall-preserving range queries.

3 HIGH-LEVEL DESIGN OF EUREKA

In this section, we provide a high-level overview of the Eureka framework, which enables efficient storage, fine-grained access,

and fast range queries on large-scale compressed scientific datasets. These datasets are typically produced once in bulk by large-scale simulations or experiments and treated as immutable, with the primary workload dominated by high-throughput reading and querying rather than frequent updates or deletions. This design assumption distinguishes EUREKA from traditional database systems, where mutability is a central concern, and allows us to prioritize compression efficiency and query performance over update support.

As shown in Figure 1, the input consists of multi-dimensional data, such as 3D grids representing temperature, pressure, or magnetic fields. The dataset is first decomposed into uniform spatial blocks, allowing each block to be compressed, decompressed, and queried independently. A customized error-bounded lossy compressor is then applied, guaranteeing reconstruction within a user-defined tolerance ε. We adopt a relative error mode, where $\varepsilon = \lambda r$ ($\lambda \in 0, 1$, r is the data range), ensuring accuracy across datasets of different scales. In practice, the choice of ε is guided by domain-specific tolerances for accuracy loss, typically determined in collaboration with domain scientists. Prior studies in areas such as climate modeling, fluid dynamics, and molecular dynamics show that error bounds are often selected based on scientifically acceptable thresholds, including permissible deviations in physical quantities or constraints for numerical stability. During compression, each block's minimum and maximum values l_j, h_j are recorded, which provide the foundation for range-based query filtering.

These ranges are organized into an AVL-tree index, where each node represents a block and stores l_j, h_j plus an augmented *max_high* attribute for pruning. We deliberately chose AVL trees over alternatives such as red–black trees or multi-dimensional structures (e.g., R-trees, K-D trees): red–black trees yield deeper hierarchies and higher memory use, while multi-dimensional indexes add unnecessary overhead for one-dimensional value-range queries. Similarly, we store only low–high ranges rather than richer statistics to keep the index compact; more complex summaries could support additional query semantics but would add significant storage and maintenance costs at scale. After construction, the tree is linearized via in-order traversal and compressed with a second-stage error-bounded scheme, preserving recall for queries, ensuring all relevant blocks are returned, with false positives filtered during decompression. Finally, the compressed data blocks and index are serialized into binary files for storage and downstream querying.

Once data and index compression are complete, EUREKA supports efficient execution of value-based range queries. As illustrated in Figure 1, when a query is submitted, EUREKA first checks whether the index is already reconstructed in memory. If so, the in-memory index is used directly to identify candidate blocks whose recorded low and high values overlap with the query range; otherwise, the compressed index is loaded from storage, decompressed, and reconstructed into the augmented AVL tree. After candidate blocks are identified, EUREKA selects the access strategy based on query selectivity. For a small number of candidates, it issues targeted reads for those blocks. When many blocks qualify, selective I/O becomes inefficient due to excessive random access; in this case, EUREKA performs a single sequential scan to load all compressed blocks but only decompresses those marked as candidates. In extreme cases where most blocks qualify, EUREKA decompresses all blocks directly to avoid performance degradation from scattered memory access.

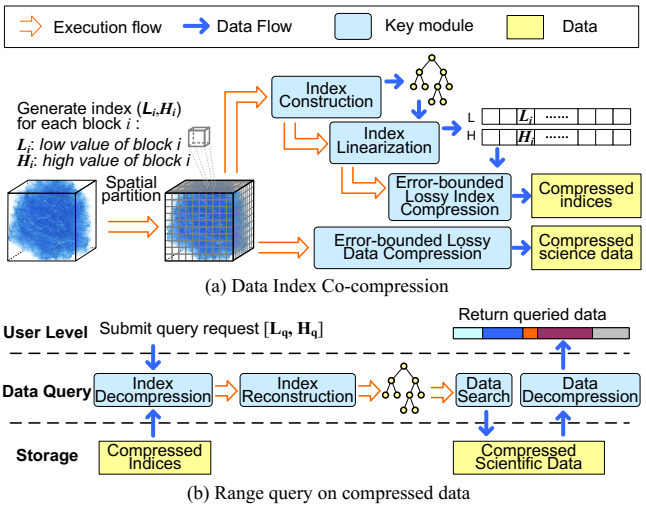

Figure 1: Overall design of EUREKA.

Finally, each candidate block is scanned at the value level to extract points within the query range, and the results are aggregated and returned to the user.

4 DETAILED DESIGN OF EUREKA

In this Section, we describe each component in EUREKA and explain the design motivation.

4.1 Domain Decomposition and Block-Level Data Compression

4.1.1 Domain Decomposition of Multi-Dimensional Scientific Data. To enable compression with fine-grained access, we first apply spatial domain decomposition to large-scale multi-dimensional datasets. In many simulation codes, field variables such as temperature, pressure, or magnetic fields are output at regular time steps as 2D or 3D grids. Since data from different time steps are stored separately, our decomposition focuses on spatial partitioning within each step, dividing each 3D grid into smaller uniform blocks.

These grids are commonly linearized into 1D arrays for efficient storage and computation. Our method operates directly on this representation: given fixed grid dimensions and block sizes, each element's array index deterministically maps to its block. As we traverse the array, values are copied into their block buffers on-the-fly, avoiding coordinate conversion or backtracking. Each element is thus processed in constant time, yielding an overall complexity of On, where n is the number of data points. In large-scale simulations that already apply domain decomposition across many processes, our method simply refines each process-local subgrid into smaller blocks. Because these subgrids are relatively small, the additional overhead is minimal and has negligible impact on performance.

4.1.2 Block-Level Data Compression. As large scientific datasets are decomposed into small blocks for fine-grained access, a key challenge is compressing each block without degrading the compression ratio. State-of-the-art compressors like SZ3 and ZFP excel

on large, contiguous arrays but perform poorly in small-block settings, which are required in our design to support efficient queries. This inefficiency arises from two factors. First, SZ3's prediction assumes strong data smoothness; on small blocks, many points lie near boundaries, reducing prediction accuracy and dispersing quantization bins. The resulting instability diminishes the effectiveness of Huffman encoding, which relies on consistent symbol frequencies. Second, SZ3 applies entropy coding independently to each block, requiring a separate Huffman tree and coding tables per block. While negligible at large scale, this metadata overhead dominates in small-block scenarios, severely undermining efficiency.

To address this, we develop a customized SZ3-based pipeline. Each block is compressed independently using a Lorenzo predictor followed by linear quantization under a user-defined absolute error bound ε, producing an integer sequence of quantized indices. Outliers beyond the quantization range are stored separately in raw form, with minimal per-block metadata recorded, while shared parameters such as ε and quantization range are stored globally once per dataset. We then replace Huffman with fixed-length bit-level encoding, which eliminates per-block metadata and directly bit-packs quantized residuals into compact binary streams. Finally, outputs are organized into three files—for outliers, sign bits, and encoded magnitudes, providing a compact layout that supports fast, selective block-level access.

4.2 Index Construction

Since compressing each block requires traversing all its elements, we record its minimum and maximum values—denoted as low and high—at no extra cost. These low–high pairs are essential for enabling value-based range queries. As shown in Figure 2, we use the low values as keys to construct an AVL tree index, where each node represents a compressed block. Although a standard AVL tree ensures logarithmic-time search, building it solely on low values limits pruning efficiency. While nodes store both low and high values, this only supports block-level overlap checks and cannot eliminate entire subtrees, reducing query performance at scale.

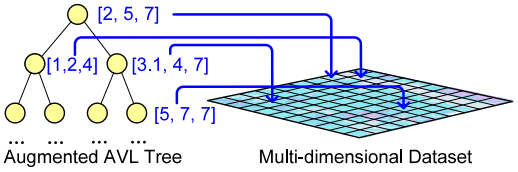

Figure 2: Illustration of index based on augmented AVL tree.

To overcome this, we augment each node with a *max_high* attribute, recording the maximum high value in its subtree. This transforms the AVL tree into a range-aware interval tree: if a subtree's *max_high* is smaller than the query's lower bound, the subtree can be skipped entirely. To avoid storage overhead, we do not store *max_high* during compression; instead, it is recomputed during index reconstruction through a bottom-up traversal. This approach eliminates redundant metadata while supporting accurate, recall-preserving range queries.

In large-scale applications, domain decomposition may yield billions of blocks, making a single AVL tree impractical due to depth and memory usage. To ensure scalability, we group blocks by spatial or temporal locality and build a separate AVL tree for each group. For example, a $1024 \times 1024 \times 1024$ grid decomposed into $4 \times 4 \times 4$ blocks can be divided into eight 512^3 subregions, each with its own tree. This reduces depth, preserves spatial locality, and improves cache efficiency. For each tree, we also record its global minimum and maximum values, enabling coarse filtering: trees whose ranges do not overlap with the query are excluded early, reducing the search space before fine-grained traversal. This hierarchical design ensures scalability and fast query responsiveness.

Table 1: Average compression ratio of fields in each dataset

Datasets	Without index	With index	CR drops by (%)
Hurricane	10.08	6.37	36.8
S3D	10.72	6.82	36.4
Miranda	16.99	8.50	50.0
NYX	9.39	5.59	40.5
WarpX	24.24	10.24	57.8
Gray-Scott	14.46	13.77	4.8

4.3 Index Compression

Indexing can introduce substantial overhead, especially when domain decomposition produces a large number of small blocks. While error-bounded lossy compressors achieve high compression ratios on raw scientific data, the overall efficiency drops significantly if indexes are stored uncompressed. As shown in Table 1, we first applied our block-level compression method to all fields of each dataset under a relative error bound of 10^{-3} and computed the resulting compression ratios, averaging across fields for each dataset. We then recalculated the ratios with uncompressed index included. The results reveal that without index compression, average compression ratios can fall by more than 50%.

To address this, we propose an end-to-end index compression pipeline for Eureka (shown in Figure 3), that effectively reduces index storage while preserving query accuracy and responsiveness.

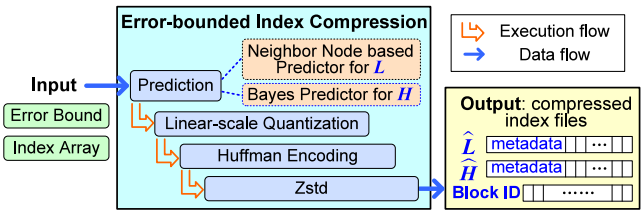

Figure 3: Index compression.

4.3.1 Index Traversing and Low Value Prediction. We exploit the structure of the AVL tree, built using block low values as keys, to enable efficient, structure-aware prediction. Because the tree preserves the sorted order of low values, a breadth-first traversal produces sequences where adjacent nodes often hold similar values. As shown in Figure 4, the first node at each level is predicted from the first node of the previous level, while subsequent nodes are predicted from their immediate left siblings. This approach captures

local correlations and keeps prediction errors small. Residuals are then quantized under an error bound (see *4.3.3*). By aligning traversal order with value ordering, this strategy enhances compression efficiency while preserving accuracy.

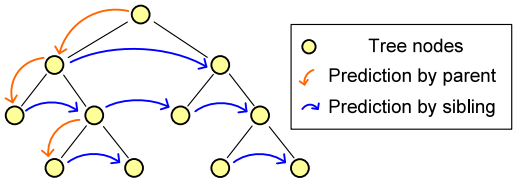

Figure 4: Illustration of low value prediction.

4.3.2 Bayesian-Based High Value Prediction. Unlike low values, high values cannot be effectively predicted using the same structure-aware traversal strategy. Since the AVL tree is built on low values, traversal produces an ordered sequence of x^{low}, enabling localized prediction with small errors. In contrast, x^{high} values are not used in tree construction and appear disordered during traversal; adjacent nodes may differ drastically, making prediction unreliable.

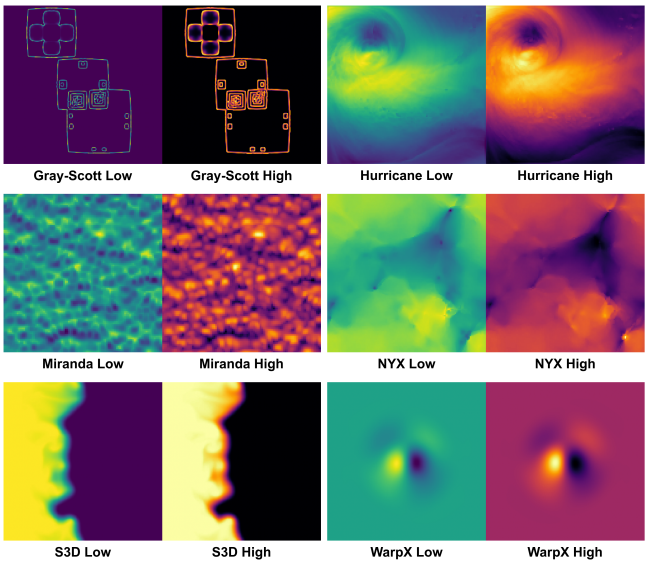

Figure 5: Visual comparison between blocks' low and high values when scientific datasets are decomposed into $4 \times 4 \times 4$ blocks.

To address this, we leverage the strong statistical correlation between low and high values within blocks. As shown in Figure 5, x^{low} and x^{high} often follow similar spatial patterns in scientific datasets. We quantify this relationship using the Pearson correlation coefficient, and as shown in Figure 6, most correlations exceed 0.9 and remain stable across time steps. This observation motivates a Bayesian-style conditional expectation model for predicting high values from low values. Specifically, we construct a joint frequency matrix $P \in \mathbb{N}^{N_{\text{low}} \times N_{\text{high}}}$ by binning normalized low and high

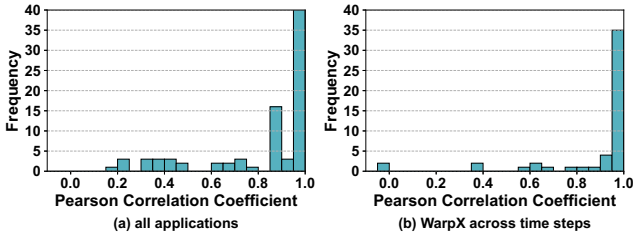

Figure 6: Pearson correlation coefficients between blocks' low and high values.

values. For each low bin l, the expected high bin index is then estimated as:

$$El = \sum_{h=0}^{N_{\text{high}}-1} \frac{Plh}{\sum_{j=0}^{N_{\text{high}}-1} Plj} \cdot h \tag{1}$$

The predicted high value $\hat{x}_i^{\text{high}}$ for a given x_i^{low} is then obtained by locating its bin index l, retrieving El, and mapping it back to the bin's center value. The full procedure is summarized in Algorithm 1.

Algorithm 1: Bayes-Based High Value Prediction

Input: Value pairs $\mathcal{S} = \{x_i^{\text{low}}, x_i^{\text{high}}\}_{i=1}^{n}$, number of bins $N_{\text{low}}, N_{\text{high}}$
Output: Predicted value array $\hat{H}1 \ldots n$
Initialize joint frequency matrix $PN_{\text{low}}N_{\text{high}} \leftarrow 0$
foreach $x_i^{low}, x_i^{high} \in \mathcal{S}$ **do**
  Map x_i^{low} and x_i^{high} to bin indices l and h
  $Plh \leftarrow Plh\ 1$ // Count co-occurrence in bins
Initialize expected bin array $E0 \ldots N_{\text{low}}-1$
foreach $l = 0$ *to* $N_{low} - 1$ **do**
  Compute El using Equation (1) // Bayesian expectation of high bin
Initialize predicted value array $\hat{H}1 \ldots n$ **foreach** $x_i^{low}, x_i^{high} \in \mathcal{S}$ **do**
  Determine low bin l from x_i^{low}
  Predict bin $\hat{h}_{\text{bin}} \leftarrow El$
  Estimate predicted value $\hat{x}_i^{\text{high}}$ as midpoint of bin $\hat{h}_{\text{bin}}$
  Set $\hat{H}i \leftarrow \hat{x}_i^{\text{high}}$
return $\hat{H}$

4.3.3 Error-Bounded Quantization. Regardless of the prediction strategy used, once the predicted value $\hat{x}_i$ is obtained, the prediction error is calculated as $\delta_i = x_i - \hat{x}_i$. This error is then quantized using a user-defined absolute error bound ε, yielding the quantization index:

$$s_i = \left\lfloor \frac{|\delta_i|}{\varepsilon} \right\rfloor$$

Values whose errors exceed the bound are treated as outliers and stored separately. To enable efficient encoding, each index s_i is combined with the sign of δ_i and offset by the quantization interval count Q, ensuring all values are non-negative. This allows signed errors to be represented compactly as non-negative integers, simplifying entropy coding and improving compression efficiency.

4.3.4 Huffman Encoding. After quantization, the indices are entropy-encoded with Huffman coding to exploit symbol frequency. A Huffman tree is built from the histogram of quantized values, assigning variable-length codes where frequent symbols get shorter codes

and rare ones get longer codes. This entropy-aware scheme minimizes the average bits per symbol, producing a compact bitstream that closely reflects the data's statistical distribution.

4.3.5 Lossless Compression of Huffman-Encoded Values. To further reduce redundancy in the Huffman-encoded bitstream, we apply Zstandard (Zstd) as a second-stage compressor. Zstd is a fast, general-purpose lossless codec that combines dictionary compression with statistical modeling, making it highly effective on large-scale index data where symbol distributions are skewed and repetitive. This two-stage pipeline, Huffman followed by Zstd, delivers high compression ratios while preserving fast decompression speeds, making it well-suited for scientific workloads that demand both compact storage and responsive query performance.

4.3.6 Lossless Compression of Block IDs. Each data block in the index is assigned a unique identifier (ID) for precise localization during queries. To minimize storage overhead, we adopt a two-level hierarchical scheme. At the higher level, the dataset is divided into coarse-grained subregions, each assigned a global subregion ID. Within each subregion, fine-grained blocks receive local IDs relative to their parent. This design narrows the numerical range of IDs, reduces bit requirements, and improves query filtering efficiency. A block's global ID can be reconstructed from its subregion and local IDs, avoiding long explicit identifiers.

For encoding, we determine the maximum possible local ID in each subregion, which specifies the minimum bit width required. Instead of fixed-width integers (e.g., 32 or 64 bits), we apply bit-packing so each ID occupies only the exact number of bits needed. The bit width is computed once per subregion and stored as metadata, ensuring constant-time decoding. This compact representation significantly reduces storage, particularly when sub-block counts per subregion are small relative to standard integer widths.

4.4 Index Decompression and Reconstruction

The index reconstruction process begins by restoring the low and high values of each AVL tree node. Low values are decompressed by reversing the pipeline: Zstandard expands the Huffman stream, which is decoded using the stored Huffman tree to recover quantization indices. Inverse quantization then yields prediction errors, which are added to predictors regenerated via structure-aware traversal, either from the first node of the previous level or the immediate left sibling, ensuring consistency and bounded error. Outliers are restored from a separate buffer. High values are reconstructed with a Bayesian estimation model. Particularly, Eureka leverages a joint frequency matrix built during compression to estimate each high value from its corresponding low bin, followed by inverse quantization. Outliers are likewise recovered from a buffer.

The tree topology is rebuilt from a level-order structure vector, which marks the presence of each node. Nodes are instantiated and connected in scan order, with low values assigned to preserve the original layout. Block identifiers are decoded from a bit-packed stream using bit-width–aware decoding, and global IDs are reconstructed from subregion and local IDs.

Once value ranges and IDs are restored, the index tree is assembled. Each node stores an approximate interval $\hat{L}, \hat{H}$, conservatively expanded to $\hat{L} - \varepsilon, \hat{H}\ \varepsilon$, as shown in Figure 7, to avoid

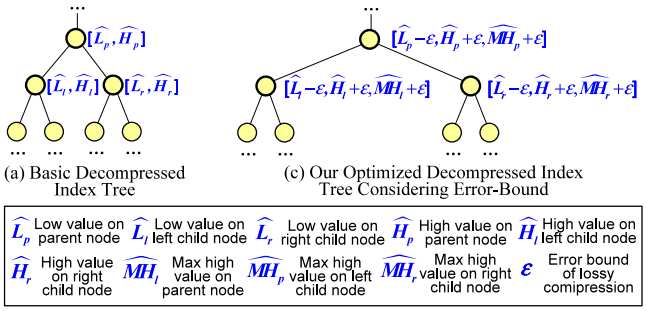

Figure 7: Decompressed AVL tree-based index.

false negatives. Any false positives are eliminated during in-block filtering. Finally, each node is augmented with a *max_high* value, the maximum $\hat{H}\ \varepsilon$ among all descendants, computed bottom-up as:

$$max\_high_p = \max\left(\hat{H}_p\ \varepsilon,\ max\_high_l,\ max\_high_r\right) \quad (2)$$

This augmentation enables efficient subtree pruning during range queries while ensuring full recall.

4.5 Range Query and Selective Data Decompression

Eureka enables efficient value-based range queries on compressed scientific datasets by leveraging the reconstructed AVL tree-based index. The query process proceeds in four steps: selecting relevant index trees, identifying candidate blocks through range search, selectively accessing and decompressing the matched data, and filtering values within the decompressed content.

4.5.1 Index Selection and Preparation. During index construction, the global dataset is partitioned into large subregions, each with its own AVL tree index built from the value ranges of its small blocks. For each subregion, the overall minimum and maximum values provide a coarse bounding interval. At query time, these intervals are first checked to identify subregions whose ranges overlap with the query interval. Only those subregions are considered further: their compressed AVL trees are loaded, decompressed, and reconstructed into balanced in-memory search trees. This hierarchical filtering avoids unnecessary decompression and narrows the search space to relevant regions.

4.5.2 Candidate Block Identification. Once the AVL tree of a selected subregion is reconstructed, it is traversed to identify small blocks whose value ranges overlap the query interval. This is achieved with a depth-first range query that uses the *max_high* attribute to prune subtrees whose ranges lie entirely outside the query bounds, as outlined in Algorithm 2. Because the AVL tree is height-balanced, the procedure runs in $\mathcal{O}\log B\ M$ time, where B is the number of blocks in the index and M is the number of matches. Compared to a linear scan, this approach visits far fewer nodes, reduces memory bandwidth usage, and improves cache locality, leading to significantly better performance.

4.5.3 Selective Block Data Access and Decompression. The block IDs identified in the previous step are then used to selectively access

Algorithm 2: Range Query with Augmented AVL Tree

Input: Decompressed AVL tree root node $\mathcal{T}$, query interval $q = L_q, H_q$
Output: Set of overlapping blocks' IDs $\mathcal{R}$
Initialize stack $\mathcal{S} \leftarrow$ and result set $\mathcal{R} \leftarrow \emptyset$
if $\mathcal{T} \neq \emptyset$ **then**
  Push $\mathcal{T}$ onto $\mathcal{S}$
while $\mathcal{S} \neq \emptyset$ **do**
  $v \leftarrow$ Pop the top node from $\mathcal{S}$
  if $v = \emptyset$ **then**
    continue
  if $v.max\_high < L_q$ **then**
    continue // The entire subtree can be safely pruned
  if $v.interval.high \leq L_q$ **and** $v.interval.low \leq H_q$ **then**
    Add v.id to $\mathcal{R}$ // Current node overlaps with the query range
  if $v.left \neq \emptyset$ **and** $v.left.max\_high \geq L_q$ **then**
    Push v.left into $\mathcal{S}$ // Left subtree should be explored
  if $v.right \neq \emptyset$ *and* $v.interval.low \leq H_q$ **then**
    Push v.right into $\mathcal{S}$ // Right subtree should be explored

and decompress the corresponding data blocks. When many blocks are involved, scattered access leads to random memory operations, which reduce cache locality, disrupt prefetching, and increase latency. Our experiments on the Expanse cluster at SDSC compared random block access with sequential all-block scanning and showed that once more than about 90% of blocks in a subregion are accessed, random access becomes as costly—or costlier—than scanning all blocks sequentially. Based on this, Eureka adopts 90% as a practical cutoff: if fewer than 90% of blocks are matched, only those candidates are decompressed; if more than 90% are matched, the entire subregion is scanned and decompressed sequentially. While the threshold may vary with hardware (e.g., HDDs, SSDs, parallel file systems), we fix it at 90% for simplicity and reproducibility. Future work will explore adaptive strategies that adjust this cutoff dynamically using hardware characteristics and runtime profiling.

4.5.4 Value Filtering within Decompressed Blocks. After the candidate blocks are decompressed and reside in memory, the final step is to perform a value-level scan within each block. The system inspects all data elements and retains only those that fall within the user-defined query interval. The result is a set of matched values along with their corresponding indices within each block, allowing for accurate downstream analysis or visualization.

4.6 Complexity Analysis

In this subsection, we present a detailed analysis of the computational and storage complexity of Eureka.

Time Complexity – Constructing Eureka requires a one-pass traversal of the dataset for block-wise compression and range collection, yielding ON time, where N is the total number of data points. The AVL-tree index is then built by inserting B blocks, costing $OB \log B$ in the worst case. Since $B \ll N$, this cost is asymptotically dominated by the ON compression step. During query processing, the worst case is ON when all blocks overlap the query. In typical low-selectivity queries, however, the AVL tree prunes irrelevant blocks in logarithmic time, giving expected query complexity $O\log B \;\; M$, where M is the number of overlapping blocks. Only these M blocks are decompressed and scanned, greatly reducing query cost compared to a full scan.

Space Complexity – The total space cost (or necessary extra memory space required) is $OQ \;\; R \;\; B$, where Q is the number of quantization bins, R is the number of outliers, and B is the number of blocks. Compression requires a Huffman tree proportional to Q, but with $Q \leq 65{,}536 \ll N$, this overhead is small. Outliers are rare (typically $\leq 1\%$ of N) and contribute negligibly. Indexing adds OB memory. Quantization bins can be generated in-place, which does not introduce extra storage. Thus, the space complexity is dominated by B and remains asymptotically far smaller than the raw dataset size ON.

5 EVALUATION

In this section, we evaluate the performance of Eureka using six representative scientific datasets on a modern HPC system.

5.1 Experimental Setup

5.1.1 Platforms. All experiments are conducted on the Expanse supercomputer at the San Diego Supercomputer Center (SDSC)[1]. Each compute node features two AMD EPYC 7742 processors (64 cores per socket, 128 threads total) and 256GB of DDR4 memory. The system runs CentOS 8, and all software is compiled with the GNU toolchain. Job scheduling is managed using SLURM. To ensure reproducibility, we report the exact software configuration: GCC 10.2.0, OpenMPI 4.1.1, and Slurm 23.02.7. The required libraries include SZ3, ZFP, ADIOS2, and zstd.

Table 2: Real-world HPC datasets used in our evaluation.

Datasets	Dims per field	Fields per step	Steps	Total Size
Hurricane [2]	500x500x100	13	1	1.3 GB
S3D [5]	500x500x500	11	1	44 GB
Miranda [3]	384x384x256	7	1	1.87 GB
NYX [4]	512x512x512	6	1	2.7 GB
WarpX [6]	1024x1024x1024	6	100	4.69 TB
Gray-Scott [1]	1024x1024x1024	2	100	1.56 TB

5.1.2 Datasets. We evaluate Eureka on six HPC simulation datasets from diverse scientific domains, summarized in Table 2. The first four datasets are part of the Scientific Data Reduction Benchmarks [42]: Hurricane (weather simulation) [2], S3D (combustion simulation) [5], Miranda (fluid dynamics) [3], and NYX (cosmology simulation) [4]. The remaining two datasets, WarpX (electromagnetic particle-in-cell simulation) [6] and Gray-Scott (reaction-diffusion simulation)[2], are generated from actual simulation runs. Both datasets contain 100 time steps and span multiple terabytes in total volume, making them representative of real-world, large-scale scientific workloads.

5.1.3 Evaluation Metrics. To evaluate the performance of Eureka in terms of compression efficiency, reconstruction quality, query performance, and scalability, we adopt four metrics.

Compression Ratio: This metric quantifies the effectiveness of compression by measuring the ratio of the original data size to the compressed size [11]. We report compression ratios for both

[1] https://www.sdsc.edu/systems/expanse
[2] https://github.com/ornladios/ADIOS2/tree/master/examples/simulations/gray-scott

the scientific data and the index structures. A higher ratio indicates more effective data reduction and improved storage efficiency.

Data Quality: Reconstruction accuracy is measured using Peak Signal-to-Noise Ratio (PSNR) [33], a standard metric for lossy compression. Higher PSNR values denote closer fidelity to the original data. This metric verifies whether Eureka maintains high reconstruction accuracy while achieving strong compression.

Query Efficiency: This metric captures the system's responsiveness to range queries, measured as end-to-end query time. It includes index lookup, block loading, decompression, and in-block value filtering. We evaluate performance under varying query selectivity and error bounds, and compare results with and without index compression to highlight the benefits of data-index co-compression.

Scalability: Scalability reflects how well the system maintains performance as dataset sizes or computational resources increase. We assess both compression and query scalability by measuring performance across varying levels of parallelism, providing insights into load balancing and resource utilization on large-scale systems.

5.1.4 Baselines. To evaluate the effectiveness of Eureka, we compare it against two categories of baselines: compression baselines and query strategy baselines.

Compression Baselines: We compare Eureka with two state-of-the-art error-bounded lossy compressors: SZ3 [41] and ZFP [25]. SZ3 is prediction-based, while ZFP uses a transform-based approach, representing the two dominant paradigms in scientific data compression [30]. For fairness, each dataset is divided into fixed-size blocks, and corresponding index structures are constructed, consistent with the block-wise design of Eureka. Each compressor is then applied independently to every block, and the index structures are also compressed. For each configuration, we separately evaluate the compression ratio of data and index, and report the overall compression ratio accordingly.

Query Strategy Baselines: To assess the benefit of index-assisted queries, we implement a baseline that applies block-wise lossy compression without indexing. Range queries are executed by sequentially decompressing all blocks and scanning for qualifying values. This represents the traditional approach for queries on compressed data and allows us to quantify the performance improvements introduced by Eureka.

5.2 Compression Ratio

To evaluate the effectiveness of Eureka in reducing storage overhead, we conducted a comprehensive compression ratio study using six representative scientific datasets under four relative error bounds ("REL" in Table 3). Each dataset includes multiple fields and time steps, and we report the minimum, maximum, and average ratios to assess robustness across fidelity requirements.

Table 3 summarizes data, index, and overall compression ratios. Eureka achieves high ratios across all datasets: over 20× at 10^{-1} for Miranda, WarpX, and Gray-Scott, and 7–20× even at 10^{-4}. Index compression is particularly strong (e.g., 294× on Hurricane) due to the regularity exploited by our predictor. Overall ratios range from 6.9× to 24.9×.

Figure 8 compares overall compression with SZ3 and ZFP. Across all datasets, Eureka consistently outperforms both baselines. For

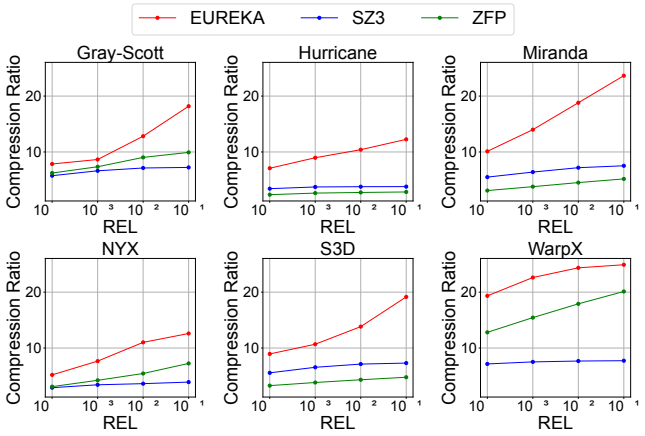

Figure 8: Overall compression ratio achieved by different compressors.

example, in WarpX with an error bound of 10^{-1}, Eureka surpasses 25×, while SZ3 and ZFP remain below 10×. This advantage becomes more pronounced with increasing error bounds, demonstrating Eureka's strong error-adaptive capability.

5.3 Data Quality

We evaluate the reconstruction quality of Eureka against state-of-the-art lossy compressors SZ3 and ZFP using rate–distortion analysis across six scientific datasets. Two standard metrics are used: (1) bit rate, the average number of bits per value in the compressed output (inversely proportional to compression ratio), and (2) Peak Signal-to-Noise Ratio (PSNR), which measures fidelity of reconstructed data—higher PSNR indicates greater accuracy.

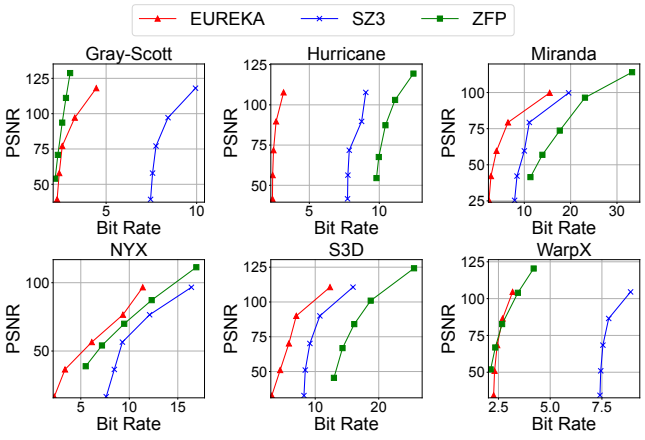

Figure 9: Rate distortion achieved by different compressors.

As shown in Figure 9, Eureka demonstrates superior performance in terms of rate-distortion across nearly all datasets. It either achieves a higher PSNR than SZ3 and ZFP at the same bit rate or attains comparable accuracy with substantially fewer bits. This trend

Table 3: Compression ratio achieved by EUREKA.

		Hurricane			S3D			Miranda			NYX			WarpX			Gray-Scott		
	REL	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Data	1E-1	9.43	13.47	12.17	19.98	20.14	20.05	16.54	26.94	25.45	13.47	13.47	13.47	26.90	26.94	26.93	14.70	26.63	20.67
	1E-2	6.50	13.47	10.95	13.95	14.23	14.08	8.75	25.84	22.07	9.04	13.47	11.84	26.03	26.42	26.3	8.70	25.33	17.02
	1E-3	4.96	13.47	10.08	10.59	10.89	10.72	5.90	21.72	16.99	4.97	13.46	9.39	22.71	25.03	24.24	5.21	23.71	14.46
	1E-4	3.56	13.46	8.89	8.80	9.08	8.93	4.74	17.48	12.24	3.27	13.43	7.08	17.39	22.65	20.85	4.87	18.71	11.79
Index	1E-1	11.15	2113.79	303.92	11.11	11.17	11.12	13.14	21.51	14.42	6.6	7.17	6.89	11.26	11.33	11.31	11.08	11.09	11.09
	1E-2	10.70	2113.79	302.66	10.77	10.86	10.81	11.94	19.69	13.47	6.37	7.17	6.81	10.95	11.29	11.16	10.94	10.96	10.95
	1E-3	10.63	2113.79	300.02	9.92	10.18	10.02	10.11	16.55	11.82	6.06	6.98	6.53	10.87	11.19	11.00	10.82	10.85	10.84
	1E-4	9.37	2113.79	294.57	8.88	9.43	9.09	7.35	10.70	8.95	5.53	6.93	6.07	6.07	10.97	9.87	10.21	10.58	10.40
Overall	1E-1	10.01	14.29	12.46	19.09	19.24	19.15	16.76	25.42	24.16	12.53	12.67	12.6	24.89	24.91	24.9	14.43	24.6	19.52
	1E-2	6.91	14.29	11.2	13.72	13.97	13.83	9.05	24.41	21.00	8.80	12.66	11.22	24.08	24.49	24.35	8.81	23.51	16.16
	1E-3	5.27	14.28	10.32	10.54	10.84	10.68	6.14	20.75	16.37	5.04	12.61	8.99	21.35	23.34	22.64	5.37	22.17	13.77
	1E-4	3.78	14.25	9.11	8.80	9.10	8.94	4.90	16.79	11.89	3.37	12.58	6.87	16.74	21.31	19.45	5.03	17.9	11.47

is observed regardless of the data domain, demonstrating EUREKA's adaptability across diverse datasets and robust performance.

While EUREKA and SZ3 both use prediction-based models and thus yield similar error-bounded accuracy, EUREKA distinguishes itself with its post-prediction encoding: a fixed-length bit-level scheme that eliminates redundancy within small blocks, producing more compact representations without loss of accuracy. Compared to ZFP's transform-based approach, which can introduce more quantization noise under irregular patterns, EUREKA shows stronger robustness and fidelity preservation, particularly at low bit rates.

Overall, the combination of accurate prediction and lightweight redundancy-minimizing encoding enables EUREKA to balance efficiency and quality more effectively than SZ3 and ZFP, achieving consistently better rate–distortion performance.

5.4 Query Efficiency

5.4.1 Impact of Index. To assess the benefits of EUREKA's index-assisted query capability, we conduct a set of performance experiments using two large-scale scientific datasets, WarpX and Gray-Scott, under varying query selectivities, defined as the proportion of values within the dataset satisfying the query condition.

We compare against a non-indexed baseline that performs block-wise lossy compression but executes range queries by sequentially reading, decompressing, and scanning all blocks. In contrast, the indexed workflow first decompresses the AVL-tree index, identifies candidate blocks via interval search, and then reads and decompresses only those blocks. Figures 10 and 11 show the results. For the baseline, query time is nearly constant across selectivities, with block decompression dominating over 90% of runtime. With indexing, query time scales with selectivity: at low selectivity, pruning irrelevant blocks yields major savings. For example, at 0.01% selectivity on WarpX, queries complete in about 4 seconds, versus over 130 seconds for the baseline—more than 30× faster.

As selectivity grows, more blocks are marked relevant, increasing both traversal and decompression costs. While indexing provides diminishing benefit in this regime, it still outperforms the baseline in most cases. An exception occurs at 10% selectivity on WarpX, where nearly all blocks are selected and poor spatial locality makes index traversal overhead outweigh filtering benefits, causing slightly slower performance than the baseline.

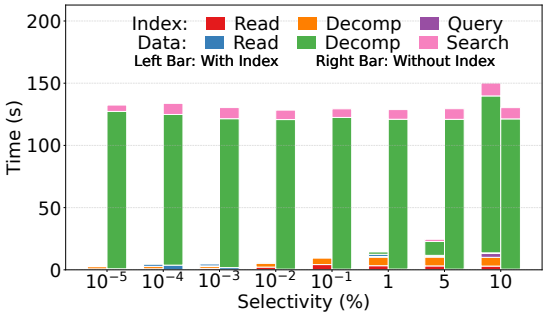

Figure 10: Query time on WarpX dataset under varying selectivity.

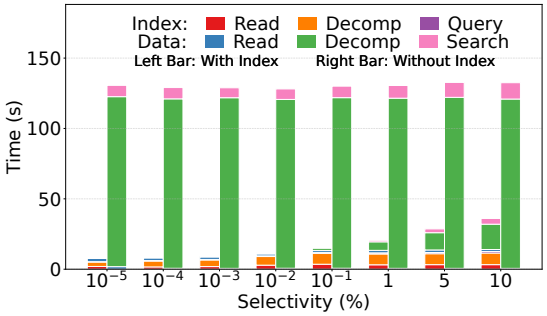

Figure 11: Query time on Gray-Scott dataset under varying selectivity.

These results confirm that EUREKA achieves substantial speedups when queries target sparse values and remains competitive at higher selectivities, validating the effectiveness of compressed indexes for reducing unnecessary data access and computation.

5.4.2 Impact of Error Bound. Building on the demonstrated effectiveness of indexing for accelerating range queries, we further examine how varying the error bound impacts query performance. This study uses two representative large-scale datasets, WarpX and Gray-Scott, under four relative error bounds: 10^{-1}, 10^{-2}, 10^{-3},

and 10^{-4}. All experiments adopt the index-assisted workflow to isolate the influence of error bounds on each query stage.

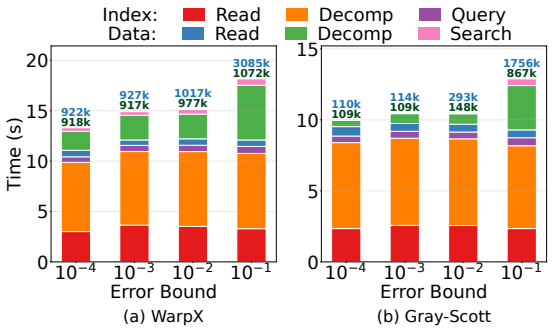

Figure 12: Query time with index under varying error bound.

As illustrated in Figure 12, total query time generally increases as the error bound loosens. Index-related operations—reading, decompression, and AVL tree traversal—remain nearly constant across error bounds, since the number of matched blocks depends only on the query range and dataset structure, not on ε. In contrast, costs for block access, decompression, and in-block filtering grow significantly with looser bounds. This is due to EUREKA's conservative error compensation: to avoid false negatives, it expands each block's recorded interval by ε, broadening the matching criteria and inflating the candidate set. Many of these blocks contain no qualifying values, yet still incur decompression overhead. This effect is especially visible in WarpX, where decompression time at 10^{-1} is nearly triple that at 10^{-4}. These results highlight that block decompression dominates query runtime and is highly sensitive to the number of candidate blocks returned by the index.

Moreover, we observe a clear decline in the precision of the filtering process as the error bound increases. Precision here is measured by the ratio of Actual Satisfactory Blocks (i.e., blocks containing values that truly satisfy the query condition) to Candidate Blocks (i.e., blocks selected by the index for further examination). This metric is visualized in Figure 12, where the top and bottom annotations on each bar indicate the number of Candidate and Actual Satisfactory Blocks, respectively. For example, in the Gray-Scott dataset, this ratio drops to about 50% under the 10^{-1} error bound, compared to nearly 99% under the tighter 10^{-4} bound. This behavior reflects a trade-off intrinsic to error-bounded indexing: larger error tolerances improve compression but reduce the selectivity of index-based filtering, resulting in higher downstream processing overhead.

5.5 Scalability Evaluation

To systematically evaluate the scalability of our framework in parallel computing environments, we divide the analysis into two parts: Preprocessing Scalability and Query Execution Scalability. These experiments assess how well the system scales with increasing numbers of MPI processes for both compression and query workloads, focusing on domain decomposition, data and index compression, and range query performance.

5.5.1 Preprocessing Scalability. The preprocessing phase includes two major tasks: domain decomposition of multi-dimensional data and the subsequent compression of both block-level data and index structures. We first evaluate the performance of domain decomposition across varying 3D grid sizes. Table 4 reports the decomposition time for datasets with number of grid points on each dimension ranging from 64 to 1024. As expected, decomposition time grows linearly with the number of grid points, i.e., On complexity.

Table 4: Decomposition time under varying data volume.

Data Volume	64^3	128^3	256^3	512^3	1024^3
Time (s)	0.03	0.24	1.96	16.64	160.87

Next, we evaluate the scalability of parallel processing during compression, focusing on two components: index processing and block-level data processing. Figure 13(a) illustrates the time required for index processing, which includes index construction and index compression. As the number of MPI processes increases from 1 to 40 (up to 4 computer nodes on Expanse are used), the overall index processing time decreases almost linearly, demonstrating excellent parallel efficiency. This result indicates that the index generation tasks are effectively distributed and incur minimal synchronization overhead.

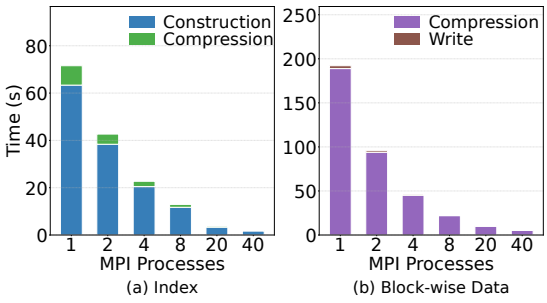

Figure 13: Compression time under varying MPI processes.

Figure 13(b) presents the time required for block-level data compression, including both the actual compression of individual blocks and writing of compressed files to disk. This stage also benefits from strong scalability. For example, using 40 MPI processes reduces the compression time from roughly 200 seconds (with a single process) to under 10 seconds, affirming the framework's ability to leverage parallel resources for efficient I/O and computation.

It is important to emphasize that both the domain decomposition and compression phases are one-time preprocessing operations. Once generated, the index structures and compressed data can be reused across multiple range queries without requiring recomputation or repeated disk access. This "build once, reuse many times" principle is essential for scalable scientific data workflows, especially when applied to large simulation outputs that must be queried repeatedly over time.

5.5.2 Query Execution Scalability. To evaluate query execution performance under varying levels of parallelism, we conduct experiments on both the indexed and non-indexed query workflows using

different MPI process counts. The results are shown in Figures 14(a) and 14(b), respectively.

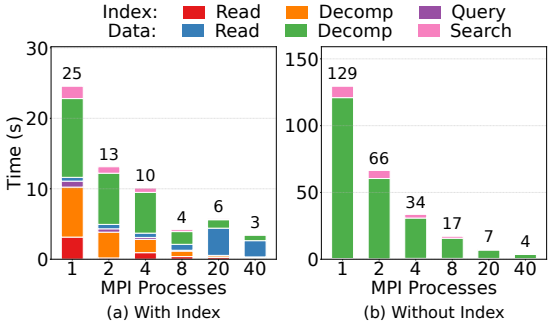

Figure 14: Query time under varying MPI processes.

In the indexed case, total query time drops significantly as more MPI processes are employed. For instance, query time decreases from 25 seconds with a single process to just 4 seconds with 8 processes, indicating effective parallel speedup. Similarly, in the non-indexed case, query time drops from 129 seconds to around 7 seconds over the same process range. The speedup in both workflows reflects the effective task distribution and data partitioning achieved by the parallel architecture.

We observe a slight increase in data reading time when scaling from 8 to 20 MPI processes, mainly due to I/O contention and bandwidth saturation under concurrent access. The impact, however, is modest, and the indexed workflow maintains its overall performance advantage.

Across all MPI configurations, the indexed approach consistently outperforms the non-indexed baseline. With only 2 processes, it achieves more than a 5× speedup, demonstrating the benefit of selective access under parallel execution. As dataset size grows, this advantage becomes even more pronounced, since non-indexed queries must always decompress and scan the entire dataset regardless of selectivity.

Overall, the framework shows strong scalability in both preprocessing and query phases, effectively leveraging parallel resources to reduce latency while keeping overhead low. These results confirm that the system is well-suited for HPC environments, offering a scalable and efficient solution for large-scale scientific data management and querying.

5.6 Comparison with DBMSs for Scientific Data

To further evaluate the performance of Eureka in scientific data management tasks, we conducted comparative experiments against two state-of-the-art database management systems (DBMSs) for scientific data: TileDB and SciDB.

TileDB adopts a tile-based storage model that partitions large multi-dimensional arrays into smaller tiles for storage and query execution. For value-range queries, however, its execution model is scan-based: tiles are located, fully decompressed, and scanned element by element. While this model works effectively for subarray queries, it is inefficient for value-range filtering because irrelevant tiles cannot be pruned prior to decompression. Prior research also highlights that tile size is a critical parameter, directly impacting metadata overhead, compression ratio, and query latency. To ensure fairness, we systematically evaluated TileDB under multiple tile-size configurations and compared its performance with Eureka under the same block-size settings.

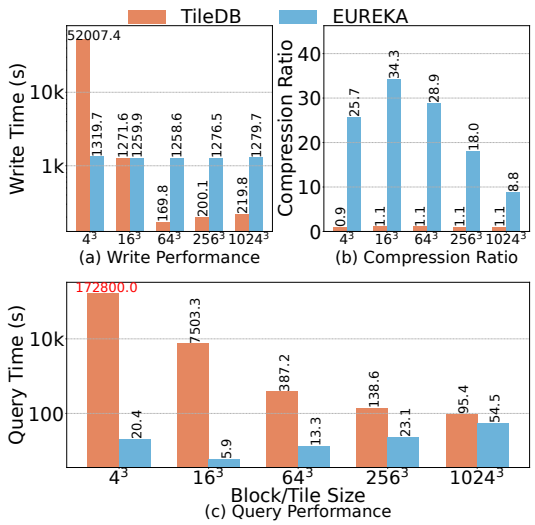

Figure 15: Performance comparison of TileDB and EUREKA.

As shown in Figure 15, TileDB's write throughput is highly sensitive to tile size, following a decreasing–then–increasing trend: very small tiles suffer from heavy metadata overhead, while very large tiles incur compression and memory penalties. In contrast, Eureka delivers stable write performance across block sizes, with cost dominated by one-time block decomposition and index construction. Although TileDB achieves slightly faster writes in its best case, this advantage is outweighed by Eureka's superior storage and query efficiency.

For compression, TileDB's ratio remains near 1.1 across configurations, while Eureka achieves far higher ratios—ranging from 8× to over 34×—demonstrating its effectiveness on large-scale scientific data. Most importantly, for query performance, Eureka consistently outperforms TileDB in low-selectivity value-range queries. With small tiles, TileDB must process vast numbers of fragments, leading to prohibitive latency (e.g., 172,800 seconds, the maximum job runtime allowed on Expanse). Larger tiles mitigate this cost somewhat, but TileDB still lags far behind Eureka, which consistently answers queries within seconds. This efficiency comes from Eureka's data–index co-compression, which prunes irrelevant blocks before decompression rather than relying on full scan-and-filter execution.

Regarding SciDB, it similarly adopts chunk-based storage and scan-based query execution, behaving much like TileDB for value-range queries. SciDB could not be deployed on Expanse due to its requirement for root privileges to run persistent database services, so we use TileDB as a representative baseline. Nonetheless, the same scan-based limitations apply to SciDB, while our results show that Eureka, by integrating lossy compression with interval indexing,

provides far more compact storage and substantially faster recall-preserving range queries at scale.

6 RELATED WORK

Scientific data management has extensively studied indexing and compression techniques, but most existing approaches treat these challenges independently.

On the indexing side, high-throughput in-memory structures such as Harmonia [38] and PACTree [23] optimize concurrency and memory access on GPUs or NVM. While excellent for transactional or streaming workloads, they prioritize latency over storage efficiency, making them less suitable for extreme-scale scientific datasets where index size and I/O dominate costs. I/O frameworks like ADIOS [36, 37] exploit lightweight per-block statistics (min/max metadata) to reduce redundant reads, and SSD-optimized designs like SIndex [34] improve block mapping and scheduling across devices. However, these methods store index metadata uncompressed, leading to substantial overhead when datasets are partitioned into millions or billions of blocks. Bitmap indexing systems such as FastBit [35] provide compact filtering for low-cardinality or categorical data [13], but they do not support hierarchical interval queries for floating-point simulation fields; discretization either harms precision or inflates index size.

Distributed spatial indexing has also been studied for scientific and geospatial workloads. Approaches such as HQ-Tree [14], OTree [10], and Hilbert R-tree variants [40] enable scalable access to multidimensional data by leveraging spatial locality. LegoIndex [19] recently extended this direction with a modular multi-level design for particle datasets. These frameworks are effective for interactive exploration and visualization, but typically rely on replication or coarse summarization, and they do not compress index structures. As a result, they incur high memory and storage costs at scale. Metadata indexing solutions such as DART [39] and BRINDEXER [28] further enrich scientific workflows by enabling efficient search on file-level metadata and hierarchical namespaces, yet they are not designed for fine-grained numerical range filtering on multi-dimensional simulation fields.

In parallel, scientific data compression has made substantial progress. Error-bounded compressors such as SZ [24, 32] and ZFP [12] achieve compression ratios of 10:1 or more by exploiting smoothness or transform-domain sparsity. GPU-accelerated schemes such as cuSZp [22] further deliver extreme throughput by adopting fixed-length encoding. However, these methods compress entire arrays and lack query-awareness; range queries still require decompressing all candidate blocks. Recent compression–query integration efforts, e.g., LeCo [26], neural time-series predictors [18], and succinct structures like FM-indexes and wavelet trees [7, 16], offer compactness and efficient point queries but still require scanning full partitions for floating-point range queries, providing no pruning of irrelevant regions.

Database management systems for scientific data, most notably TileDB [8] and SciDB [31], have adopted array-based storage and tiled indexing to enable scalable analysis. These systems excel at subarray queries and spatial filtering but still require decompressing and scanning all tiles that overlap the query range. As our evaluation shows, TileDB's value-range query latency remains orders of magnitude higher than Eureka, and its compression ratios hover near 1.1, far below what is required for petascale and exascale workloads.

In contrast, Eureka addresses these limitations through a co-design of compression and indexing. By organizing data into small blocks, building AVL-tree-based interval indexes, and compressing both data and index with multi-stage schemes, Eureka enables recall-preserving pruning of irrelevant blocks before decompression. This reduces storage overhead, avoids unnecessary I/O, and accelerates value-range queries. Unlike existing indexing systems, which store large metadata structures uncompressed, and unlike lossy compressors, which lack query-awareness, Eureka integrates both dimensions into a unified, scalable framework tailored for large-scale HPC workflows.

7 CONCLUSION AND FUTURE WORK

This paper presents Eureka, a unified data–index co-compression framework that enables efficient fine-grained access and value-range queries on large-scale scientific datasets. By integrating spatial domain decomposition, block-wise error-bounded compression, and hierarchical AVL-tree indexing, Eureka simultaneously reduces data and metadata storage while maintaining high query fidelity. Unlike prior approaches that treat compression and indexing separately, Eureka compresses both components within a coordinated pipeline, producing compact, query-aware storage with bounded-error reconstruction. Experiments on six diverse HPC simulation datasets show that Eureka achieves up to 25× data compression and over 300× index compression, outperforming state-of-the-art compressors (SZ3, ZFP) in rate–distortion tradeoffs. Its index-assisted query engine further delivers over 30× speedup on low-selectivity range queries by avoiding unnecessary decompression and I/O. Eureka also scales efficiently in both preprocessing and query execution, exhibiting near-linear speedups with increased parallelism—making it well suited for HPC environments where data volume and access latency are critical constraints.

In future work, we plan to extend Eureka with GPU acceleration to further reduce compression and query latency, and explore alternative indexing structures (e.g., B+-trees, R-trees) to support a wider range of data distributions and analysis patterns. While dynamic updates are not currently supported, emerging workloads such as streaming sensor data and iterative simulations may require partial updates. We therefore aim to develop efficient update mechanisms that preserve Eureka 's compression effectiveness and query performance.

ACKNOWLEDGMENTS

This research was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under Contract No. DE-AC02-06CH11357. Additional support was provided by the National Science Foundation under Grant Nos. 2531903, 2311875, 2344717, and 2514036. This work was also supported by the U.S. Army Research Laboratory under Cooperative Agreement No. W911NF-23-2-0224.

REFERENCES

[1] [n. d.]. Gray-Scott Simulation in ADIOS2. https://github.com/ornladios/ADIOS2/tree/master/examples/simulations/gray-scott Accessed: 2024-11-08.

[2] [n. d.]. Hurricane ISABEL simulation dataset in IEEE Visualization 2004 Test. http://vis.computer.org/vis2004contest/data.html Accessed: 2024-11-03.

[3] [n. d.]. Miranda: Compressible Multiphysics Simulation. https://wci.llnl.gov/simulation/computer-codes/miranda Accessed: 2024-10-03.

[4] [n. d.]. NYX simulation. https://amrex-astro.github.io/Nyx/ Accessed: 2024-12-06.

[5] [n. d.]. S3D: Direct Numerical Simulation of Turbulent Combustion. https://crf.sandia.gov/research/computation-and-theory/direct-numerical-simulation/ Accessed: 2024-11-03.

[6] [n. d.]. WarpX: A Highly-Parallel Electromagnetic Particle-In-Cell Simulation. https://github.com/BLAST-WarpX/warpx Accessed: 2024-11-08.

[7] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A learned approach to design compressed rank/select data structures. *ACM Transactions on Algorithms (TALG)* 18, 3 (2022), 1–28.

[8] Jacob Bolewski and Stavros Papadopoulos. 2017. Managing massive multi-dimensional array data with TileDB:—Invited demo paper. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 3175–3176.

[9] DJ Campbell. 2001. The physics of the international thermonuclear experimental reactor FEAT. *Physics of Plasmas* 8, 5 (2001), 2041–2049.

[10] Cesare Cugnasco, Hadrien Calmet, Pol Santamaria, Raül Sirvent, Ane Beatriz Eguzkitza, Guillaume Houzeaux, Yolanda Becerra, Jordi Torres, and Jesus Labarta. 2019. The OTree: Multidimensional indexing with efficient data sampling for HPC. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 433–440.

[11] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 730–739.

[12] James Diffenderfer, Alyson L Fox, Jeffrey A Hittinger, Geoffrey Sanders, and Peter G Lindstrom. 2019. Error analysis of ZFP compression for floating-point data. *SIAM Journal on Scientific Computing* 41, 3 (2019), A1867–A1898.

[13] Bin Dong, Surendra Byna, and Kesheng Wu. 2014. Parallel query evaluation as a scientific data service. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 194–202.

[14] Jun Feng, Zhixian Tang, Mian Wei, and Liming Xu. 2014. HQ-Tree: A distributed spatial index based on Hadoop. *China communications* 11, 7 (2014), 128–141.

[15] Jeffrey P Freidberg. 2008. *Plasma physics and fusion energy*. Cambridge university press.

[16] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*. Springer, 326–337.

[17] Richard Grotjahn, Robert Black, Ruby Leung, Michael F Wehner, Mathew Barlow, Mike Bosilovich, Alexander Gershunov, William J Gutowski, John R Gyakum, Richard W Katz, et al. 2016. North American extreme temperature events and related large scale meteorological patterns: a review of statistical methods, dynamics, modeling, and trends. *Climate Dynamics* 46 (2016), 1151–1184.

[18] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned compression of nonlinear time series with random access. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 1579–1592.

[19] Chang Guo, Ning Yan, Lipeng Wan, and Zhichao Cao. 2025. LegoIndex: A Scalable and Modular Indexing Framework for Efficient Analysis of Extreme-Scale Particle Data. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*. 1–14.

[20] Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. 2017. Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 1–8.

[21] Tony Hey, Stewart Tansley, Kristin Michele Tolle, et al. 2009. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA.

[22] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuszp: An ultra-fast gpu error-bounded lossy compression framework with optimized end-to-end performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[23] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 424–439.

[24] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.

[25] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[26] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. Leco: Lightweight compression via learning serial correlations. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–28.

[27] Sarah E Marzen and Simon DeDeo. 2017. The evolution of lossy compression. *Journal of The Royal Society Interface* 14, 130 (2017), 20170166.

[28] Arnab K Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R Butt. 2020. Efficient metadata indexing for hpc storage systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 162–171.

[29] Philip J Rasch, S Xie, P-L Ma, W Lin, H Wang, Q Tang, SM Burrows, P Caldwell, K Zhang, RC Easter, et al. 2019. An overview of the atmospheric component of the Energy Exascale Earth System Model. *Journal of Advances in Modeling Earth Systems* 11, 8 (2019), 2377–2411.

[30] Seung Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Weikeng Liao, and Alok Choudhary. 2014. Data Compression for the Exascale Computing Era - Survey. *Supercomput. Front. Innov.: Int. J.* 1, 2 (jul 2014), 76–88. https://doi.org/10.14529/jsfi140205

[31] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A database management system for applications with complex analytics. *Computing in Science & Engineering* 15, 3 (2013), 54–62.

[32] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1129–1139.

[33] Dingwen Tao, Sheng Di, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2019. Z-checker: A framework for assessing lossy compression of scientific data. *The International Journal of High Performance Computing Applications* 33, 2 (2019), 285–303. https://doi.org/10.1177/1094342017737147

[34] Shucheng Wang, Kaiye Zhou, Zhandong Guo, Qiang Cao, Jun Xu, and Jie Yao. 2024. SIndex: An SSD-based Large-scale Indexing with Deterministic Latency for Cloud Block Storage. In *Proceedings of the 53rd International Conference on Parallel Processing*. 1237–1246.

[35] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: Interactively Searching Massive Data. Lawrence Berkeley National Lab. (LBNL), Berkeley, CA (United States). https://www.osti.gov/biblio/964373 Accessed: 2024-11-06.

[36] Tzuhsien Wu, Jerry Chou, Shyng Hao, Bin Dong, Scott Klasky, and Kesheng Wu. 2017. Optimizing the query performance of block index through data analysis and I/O modeling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10.

[37] Tzuhsien Wu, Jerry Chou, Norbert Podhorszki, Junmin Gu, Yuan Tian, Scott Klasky, and Kesheng Wu. 2017. Apply block index technique to scientific data analysis and I/O systems. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 865–871.

[38] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: a high throughput B+ tree for GPUs. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 133–144.

[39] Wei Zhang, Houjun Tang, Suren Byna, and Yong Chen. 2018. DART: distributed adaptive radix tree for efficient affix-based keyword search on HPC systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–12.

[40] Yu-Hang Zhang, Chang Wen, Min Zhang, Kai Xie, and Jian-Biao He. 2022. Fast 3D visualization of massive geological data based on clustering index fusion. *IEEE Access* 10 (2022), 28821–28831.

[41] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[42] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific data reduction benchmark for lossy compressors. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2716–2724.