ABSTRACT

LAKSHMINARASIMHAN, SRIRAM. Enabling Query-driven Analytics via Extreme Scale In-situ Processing. (Under the direction of Nagiza F. Samatova.)

Efficient analytics of scientific data from extreme-scale simulations is quickly becoming a top-notch priority. Scientific simulations that are driven by local space-time relationships are largely performed with the purpose of discovering or explaining non-local and large-scale space-time relationships through interactive, query-driven, what-if data exploration. The data generation process of typical simulations proceed from one time step to the next and requires the context of only two time steps, while storing data for only one time step on the disk. In contrast, data analytics often requires the full context of the available data, not just a single time step. Thus, the fundamental differences in data context and heterogeneity of access patterns demand analytics-driven data management solutions.

One promising approach for achieving these ends is in-situ processing, i.e., processing simulation output while the output still resides in applications’ memory. However, enabling in-situ data-analytics at extreme-scale requires a number of algorithmic challenges to overcome. First, coupling analysis with simulation execution requires techniques that are non-intrusive to the simulation, while being computationally efficient and having the ability to operate in-core. This constraint renders many current state-of-the-art analytics techniques inapplicable in an in-situ context. Second, data-intensive analysis routines, which are predominantly query-driven, are bottlenecked on access to slow storage devices. Thus, to sustain reduced response times over extreme scale data, analytics routines must breakaway from the existing practice of creating large pre-computed indexes that trade storage for computational efficiency. Third, before the simulation output leaves the compute cluster, data must first be made analysis-ready, by aggregating across spatial/temporal resolutions. This however, needs to happen in-network, under network-bandwidth and memory constrained scenarios. Arguably, a transformative shift is necessary, making data analytics and data reduction first class citizens of the data management design and information processing.

In this thesis, we propose a new approach that enables query-driven analytics over compressed scientific data sets—rather than original, full-size data—, in order to meet storage-and-I/O-bound application challenges. We first present an efficient data compres-
sion routine in In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) of scientific data that is unlike existing lossless compression techniques, which are hardly suitable for scientific data due to its inherently entropic nature. With ISABELA, we apply a **preconditioner** to seemingly random and noisy data along spatial resolution to achieve an accurate fitting model that achieves a $\geq 0.99$ correlation with the original data while providing compression rates of $\approx 85\%$ and introducing only a negligible overhead on simulations in terms of runtime.

To support analytics-driven efficient query processing, we build a parallel query processing engine, called ISABELA-QA. ISABELA-QA takes an orthogonal approach when compared with the traditional indexing of raw data, by binning and generating a value-based index over ISABELA-compressed data that results in an order of magnitude reduction in index storage. The memory light-weight index enables a query execution model that offers more than a 10-fold speed-up in per-core processing along with a multi-fold reduction in energy consumption. By turning an IO-bound problem into a compute-bound problem, we achieve a design that is inherently scalable on multi-node, multi-core, and GPU configurations.

And finally, we propose DIRAQ, a parallel **in-situ, in-network** data indexing and reorganization technique that enables the transformation of simulation output into a query-efficient form, with negligible runtime overhead to the simulation run. DIRAQ begins with an effective core-local encoding, which comprises a highly-compressed inverted index that results in a total storage footprint less than the raw data. DIRAQ then applies an **in-network** index merging strategy, enabling the creation of aggregated indexes ideally suited for analytical queries that speed up query responses by up to 10x versus alternative techniques. We also employ a novel aggregation strategy that is topology-, data-, and memory-aware, resulting in efficient I/O and yielding overall end-to-end encoding and I/O time that is up to 6x faster than that required to write the raw data with MPI collective I/O.
Enabling Query-driven Analytics via Extreme Scale In-situ Processing

by
Sriram Lakshminarasimhan

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science

Raleigh, North Carolina 2013

APPROVED BY:

_________________________  _______________________
Xiaosong Ma                Frank Mueller

_________________________  _______________________
Kemafor Ogan               Nagiza F. Samatova
Chair of Advisory Committee
DEDICATION

In loving memory of my father, S. Lakshminarasimhan, and grandfather, G. N. Ramabhadran.
BIOGRAPHY

Sriram Lakshminarasimhan earned his Bachelor of Engineering degree in Computer Science and Engineering in 2007 from Anna University, India, graduating with a distinction. He then worked as a software developer in India for two years, before entering North Carolina State University in Fall 2009 to pursue graduate studies in Computer Science. During his doctoral study, he has spent summers at VMWare and Argonne National Laboratory. He passed his Written Qualifier Examination in Summer 2011 and Preliminary Oral Examination in Spring 2013.
ACKNOWLEDGEMENTS

The successful completion of this dissertation would not have been possible without the goodwill and guidance of several people.

First and foremost, I am indebted to my advisor Nagiza Samatova for taking me under her wing and guiding me throughout my thesis study. It was her encouragement in the first place that made me convert to the Ph.D. program. Her confidence in me has on several occasions been overwhelming, but it has directly contributed to my growth as a person. Her research productivity, determination and drive have never ceased to amaze me and these characteristics have been a constant source of motivation for me to improve my own work ethic. Even with all my efforts, the primary accelerators for the timely and successful completion of this dissertation have been Nagiza’s open-door policy and round-the-clock reachability. I could go on, but keeping it short, I feel truly blessed to have an advisor like her, and, not surprisingly, her insight and guidance have extended well beyond my academic life.

I am very thankful to my committee members: Professors Xiaosong Ma, Frank Mueller and Kemafor Ogan, for their valuable support, guidance and constructive comments that have helped improve this thesis. I am grateful to Douglas Reeves for admitting me into the Ph.D. program with funding. I had the privilege to learn a great deal and derive inspiration from several professors in the department, particularly Rada Chirkova and George Rouskas.

I have learned a lot from collaborating with several experts in the field of high-performance computing: Scott Klasky, Norbert Podhorszki and Qing Liu from Oak Ridge National Laboratory and Rob Ross, Venkat Vishwanath and Rob Latham from Argonne National Laboratory. I am especially thankful to them for providing me access to leadership class computing resources and the necessary software infrastructure for me to work with. The summer I spent at Argonne, with Venkat Vishwanath, Mark Hereld and Michael Papka, was particularly productive, and I am grateful to them for providing such an excellent research environment. Additionally, I would like to thank Stephane Ethier, Sueng-Hoe Ku, C.S. Chang, Hemanth Kolla and Jackie Chen for providing me access to, and helping me understand the simulation datasets I have used throughout my thesis.

This thesis would not have been complete without the help from research mates here at NCSU. I am deeply thankful to John Jenkins and Neil Shah for taking the time to
critique my papers and offer corrections that markedly improved the presentation of the papers. I am thankful to David (Drew) Boyuka for the work on DIRAQ. I have enjoyed innumerable long discussions with him on a wide variety of topics, which have almost always ended up being enlightening for me. Eric Schendel deserves a special mention for being the go-to guy in our group on addressing any questions regarding code optimization. I am grateful to Isha Arkatkar, Zhenhuan Gong, Xiaocheng Zou and Saurabh Pendse for assistance on various papers.

I am under immense obligation to Kanchana Padmanabhan – my confidante, counsel and closest friend for the past 4 years. Her positive attitude and patience have been a source of inspiration for me. And special thanks to her for sitting through multiple dry-runs of almost all my presentations and offering constructive criticisms and suggestions that helped me polish them. I would like to thank my friends Naga, Sairam and TC for flying in for my defense. In addition to them, I would like to thank Mohit, Ramprasad, Sandeep, Suresh, Vasanth, and Vignesh for their constant support and encouragement.

Finally, none of this would have been even remotely possible if it were not for the patience and support of my family members. My mother Chandralekha, sister Vidya and brother-in-law S.K. Rangaswamy have been pillars of strength and their sacrifices have made my Ph.D. study realizable. I would also like to express my gratitude to my relatives Ravi and Jayashree for providing me with the financial assistance that helped me get started with my graduate studies.

The work of Sriram Lakshminarasimhan was funded by the U.S. Department of Energy, Office of Science (SciDAC SDM Center) and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).
# TABLE OF CONTENTS

## LIST OF TABLES

- Chapter 1 Introduction
  - 1.1 Evolving Data Analysis Pipeline
    - 1.1.1 In-situ Processing for Query-driven Analytics
  - 1.2 Challenges with Storage-Efficient Scientific Data Analytics
    - 1.2.1 Compression
    - 1.2.2 Indexing
  - 1.3 Hypothesis
  - 1.4 Proposed Approaches
    - 1.4.1 ISABELA for Effective In-situ Compression of Scientific Data
    - 1.4.2 ISABELA-QA: Query-driven Analytics over ISABELA-compressed Data
    - 1.4.3 DIRAQ: Scalable Data Encoding for Analytical Query Processing

## LIST OF FIGURES

- Chapter 1 Introduction
  - 1.1 Evolving Data Analysis Pipeline
  - 1.2 Challenges with Storage-Efficient Scientific Data Analytics
  - 1.3 Hypothesis
  - 1.4 Proposed Approaches

- Chapter 2 ISABELA for Effective In-situ Compression of Scientific Data
  - 2.1 Introduction
  - 2.2 A Motivating Example
  - 2.3 Problem Statement
  - 2.4 Theory & Methodology
    - 2.4.1 Sorting-based Data Transformation
    - 2.4.2 Background: Cubic B-splines Fitting
    - 2.4.3 Maximizing Compression Ratio via Window Splitting
    - 2.4.4 Error Quantization for Guaranteed Point-by-point Accuracy
    - 2.4.5 Exploiting ∆-encoding for Temporal Index Compression
    - 2.4.6 ISABELA Data Workflow
  - 2.5 Results
    - 2.5.1 Per Window Accuracy
    - 2.5.2 Effect of Window Size $W_0$ on Accuracy
    - 2.5.3 Trade-off between Compression and Per Point Accuracy
    - 2.5.4 Effect of ∆-encoding on Index Compression
    - 2.5.5 Compression Time
    - 2.5.6 Performance for Fixed Compression
    - 2.5.7 Performance for Fixed Accuracy
    - 2.5.8 Scientific Data Analysis on ISABELA-compressed Data
  - 2.6 Related Work
  - 2.7 Conclusion
**LIST OF TABLES**

| Table 2.1 | Summary of GTS output data by different categories. | 16 |
| Table 2.2 | Performance of examplar lossless and lossy data compression methods. | 17 |
| Table 2.3 | Impact of $\Delta$-encoding on $CR$ for Potential (Density). | 29 |
| Table 2.4 | ISABELA vs. Wavelets for fixed $CR = 81\%$ and $W_0 = 1,024$. | 30 |
| Table 2.5 | XGC data analysis showing the correlation between radial zones on original data. | 33 |
| Table 2.6 | Impact of error quantization on correlation between radial zones on ISABELA-compressed data, and the difference with the correlation over original data. | 33 |
| Table 2.7 | Mis-classification rate for 100-iteration $k$-means on the Flash and GTS datasets. Sample size is 4096 2-D points. $\epsilon$ represents the bounded error per data point. | 35 |
| Table 2.8 | S3D Global maxima of absolute relative errors with the original data, for mean and standard deviation across 50 timesteps for 20 million points. For mean of velocities $u$, $v$, $w$, we consider values $> 3m/s$, and for standard deviation $\sigma > 1m/s$. For temperature we consider values with $\sigma > 5$ Kelvin (the peak variance in the domain is about $3.8E + 05K^2$). | 36 |
| Table 4.1 | Effect of dataset entropy (number of bins) on index compression ratio and total size on 256 processes each indexing 1 MB of data on BG/P. | 71 |
| Table 4.2 | Parameters for the performance model. | 79 |
LIST OF FIGURES

Figure 1.1 The scientific method cycle of today’s extreme-scale simulation applications. ................................................. 2
Figure 1.2 Components of this thesis. .................................................. 10
Figure 2.1 A slice of GTS Potential: (A) original; (B) sorted; (C) decoded after B–splines fitting to original; and (D) decoded after B–splines fitting to sorted. .................................................. 18
Figure 2.2 A cubic B-spline fitting with $m = 6$ control points, $k = 10$ knots, and with 3 piecewise cubic segments. .................. 20
Figure 2.3 Illustration of $\Delta$-encoding of the index across temporal resolution. .............................................................. 24
Figure 2.4 Workflow of ISABELA compression engine from data generation to in-situ compression to storage. .......................... 25
Figure 2.5 Accuracy ($\rho$): (a) Per window correlation for Wavelets, B–splines, and ISABELA with fixed $CR = 81\%$ for GTS Density. (b) Per window correlation for GTS linear and non-linear stage Potential decompressed by ISABELA. .................................................. 26
Figure 2.6 Sensitivity of NRMSE values for ISABELA-compressed GTS Potential data across 100 windows over varying window sizes. The compression rate is fixed at $CR = 81.44\%$ .................................................. 27
Figure 2.7 Compression ratio ($CR$) performance: (a) For various per point relative error thresholds ($\tau_\epsilon$) in GTS Potential during linear and non-linear stages of the simulation. (b) For various timesteps with $\tau_\epsilon = 1\%$ at each point (for GTS Potential: $t_1 = 1,000, \Delta t = 1,500$; for Velocity in Flash: $t_1 = 3,000, \Delta t = 3,500$). .................................................. 28
Figure 2.8 Compression ratio ($CR$) performance with per-window constraint $\rho > 0.99$, and NRMSE < 0.01: (a) On each window, with varying number of coefficients, in Flash velx (b) Overall storage cost over 400 windows across various petascale simulation datasets. .... 31
Figure 2.9 Comparison of intensity plots of the normalized temperature values in XGC data, across temporal and poloidal dimensions for original (left) and ISABELA (middle) decompressed data. The absolute difference between the derived values, on an order of magnitude lower scale, are shown in the figure on the right. ................. 32
Figure 2.10 Plot of derived turbulence intensity values over 620 timesteps across different radial zones in XGC simulation. .............. 34
Figure 3.1 Overview of the query processing engine. Values in brackets represent optional pieces of the flow, such as variables being reconstructed for bins in $B_A$ when $V_q \neq \emptyset$. See Section 3.3. .................. 45
Figure 3.2 Bin-based index file format for spatial regions.  
Figure 3.3 Bin-based index file formats for quantized errors. Each window tuple contains the window ID, the window offset, the number of values within the window in the bin, and the offset in memory of the uncompressed $k$-vector. *Each section is compressed using the standard compression library zlib.  
Figure 3.4 Comparison of the accuracy for GTS Potential data for FastBit, between quantized and raw values with level of binning. Significant relative error $\epsilon > 1\%$, is relative error ($\epsilon$) of a quantized value with respect to its original value.  
Figure 3.5 Precision of query results obtained on ISABELA-compressed 3.2GB GTS potential data.  
Figure 3.6 Comparison of end-to-end query response times over varying selectivity on data of size 1 TB.  
Figure 3.7 Comparison of end-to-end region-centric query response times over varying selectivity on data of size 32 GB.  
Figure 3.8 Performance of computational components of ISABELA-QA query processing over varying selectivity.  
Figure 3.9 Comparison of overall compute and I/O costs over varying selectivity.  
Figure 3.10 A comparison between serial, CPU-parallelized, and GPU-parallelized B-spline regeneration. 8 OpenMP threads were used in this particular plot, corresponding to two quad core Intel Xeon X5355 processors.  
Figure 3.11 Speedup of queries requesting spatial regions only and optionally variables. 16 OpenMP threads are used, on four quad-core processors on a single node.  
Figure 3.12 Components of distributed query processing over larger (10%) selectivity sizes of the 32 GB GTS data.  

Figure 4.1 Overview of the DIRAQ indexing methodology, with lightweight encoding$^{1,2}$, scalable index aggregation$^{8,9}$ and resource-aware dynamic aggregator selection$^6$.  
Figure 4.2 An overview of how raw floating-point data is encoded by DIRAQ for compact storage and query-optimized access. CII stands for compressed inverted index.  
Figure 4.3 A logical view of group index aggregation and writing.  
Figure 4.4 Steps for building a group index layout on the leader core (shown for a group size of 2, generalizable for more cores).  
Figure 4.5 Accuracy of the performance model prediction on Group Layout, Aggregation, and I/O timings with compression for a fixed compute to aggregator ratio of 8:1. A, P stand for actual and predicted timings, respectively.
Figure 4.6  Comparison of response times of DIRAQ compressed (CII) and uncompressed (II) indexes with FastQuery, over various aggregation sizes, on queries of fixed-selectivities. ................................. 82

Figure 4.7  Resulting index sizes (as a % of raw data) on varying amount of data aggregated per-core with FastQuery (FQ) and DIRAQ (CII) indexing techniques. .......................................................... 83

Figure 4.8  Strong scaling on the effective end-to-end throughput (original data size / end-to-end encoding time) on 3 FLASH simulation datasets indexing a total of 2 GB, on Intrepid. PFPP (POSIX file-per-process) and MPI-IO (two-phase collective) perform raw data writes with no overhead of indexing. .......................................................... 84

Figure 4.9  Weak scaling on the effective throughput (original data size / end-to-end indexing time) for 3 FLASH simulation datasets indexing 1 MB on each core on Intrepid. PFPP, MPI-IO performs raw data writes with no overhead of indexing. ................................. 85

Figure 4.10 Cumulative time spent on each stage of DIRAQ for 4 datasets from FLASH simulation on Intrepid, with each process indexing 1 MB of data using an uncompressed index (II), and a compressed inverted index (CII). .......................................................... 86

Figure 4.11 Comparison of different aggregation strategies (number of compute to aggregator process) on I/O and aggregation timings, with process ranks sorted by aggregation times. 1 MB of the dataset vely is indexed by each of the 4096 processes. ................................. 88
Chapter 1

Introduction

Scientific simulations in the past decade have utilized the exponential growth in computational capabilities, effectively scaling to hundreds and thousands of processes. As a result, these simulations can now simulate physical phenomena at unprecedented level of detail, scale and can generate up to several hundreds of terabytes of data during a single simulation run. However, with increasing amounts and complexity of data produced, post-processing analysis is becoming increasingly challenging, and without a proportionate increase in I/O bandwidth, this problem is expected to get exacerbated.

Analysis and visualization of large-scale simulation data sets place unique I/O requirements owing to fundamental differences in data context and access patterns among the simulation and exploratory data analytics steps (see Figure 1.1). The data generation process of space-time simulation proceeds in a local manner, from one time step to the next, and requires the context of only two time steps, while storing data for only one time step at a time on the disk. In contrast, data analysis and visualization steps often require the full context of the available data, not just a single time step. In fact, simulations that are driven by local space-time relationships are largely performed with the purpose of discovering or explaining global and large-scale space-time relationships through analysis and visualization [28].

The full context analysis and visualization software become tremendously data-intensive, creating a potential bottleneck in the entire scientific discovery cycle. Scaling up to full context faces severe constraints of computer memory, forcing the results of multiple data analysis and visualization steps to be stored on the disk. Moreover, such a data exploration process is interactive in nature, requiring almost real-time I/O access rates. Thus,
the analysis and visualization software rely on high-performance I/O much more heavily than simulations in order to provide scientists with an interactive environment for the full context data exploration. However, it is a well-known phenomenon that I/O access rates have not kept pace with high-performance computing performance as a whole. For this reason, new paradigms are needed to support the unique needs of analysis and visualization for extreme-scale simulation applications, especially in the full context.

Typically, exploratory accesses over scientific datasets are made in the form of queries that involve constraints on one or more variable values. For example, a commonly used query over S3D combustion simulation [34] datasets involves identifying all regions in space at a particular timestep where \( 100 < uvel < 120 \) and constructing a probability density function or a histogram of the values satisfying the points of interest. Based on the analysis of the above output, a new query may be formulated, where either the query ranges or the variables themselves are updated. And, as this process is repeated iteratively, constant access on the entire raw output over slow devices quickly becomes inefficient.

Rather than performing a full scan of the data for each incoming query, utilizing an efficient indexing scheme can help accelerate query response times by several orders of magnitude. Indexing techniques such as bitmap indexes [6, 71] and B-Trees [18] have shown to extremely effective in a large number of scenarios involving databases and also with scientific applications [52, 74]. However, at current rates of data generation that are seen with extreme-scale simulations, the storage overhead of these indexing techniques can be more than 200% of the data being indexed. While indexes are traditionally designed to trade storage for computational efficiency, application scientists’ operate under environments that are constrained on both limited I/O storage and access bandwidth\(^1\). Hence, generally, application scientists are not interested in solutions that result in more

---

\(^1\)The IBM BG/P cluster “Intrepid” at Argonne National Laboratory that is equipped with 163,840 compute cores can deliver compute performance of 0.5 TeraFlops/second, but the I/O performance is limited to a few GBs/second as the operating filesystem is 97% full as of writing.
data [60] and, in contrast, reducing data and its movement has become a top priority at exascale.

1.1 Evolving Data Analysis Pipeline

The scientific data analyses have been evolving from a pure post-processing regime to one with a hybrid mixture of simulation-time and post-simulation form of analysis. Two main factors have influenced this change in the analysis pipeline. First, simulation-time analysis eliminates the need for writing the data to storage and reading it back for processing by storing the data to be analyzed in memory of analysis or compute nodes. In contrast, post-processing (processing the output after simulation completion) has become an increasingly tedious process, where extensive movement of terabytes and petabytes of data to and/or from storage nodes is necessary and highly expensive. Second, simulation-time processing not only allows immediate insight into the output, but also provides a framework that can equip scientists with the ability to explore data at real-time. The vision of such a framework involves feeding results of analyses back into the running simulations and eventually steering them [47]. [It is noted that not all analysis routines can be performed concurrently with the running simulation.] Despite an increase in the number of compute resources, the available memory is insufficient to aggregate and analyze data across a large number of timesteps. Large-scale time-series analyses will still need to be deferred till post-simulation in order to perform out-of-core processing. However, several forms of data exploration and a large class of analysis routines can take advantage of the improved compute capabilities on leadership class machines, making simulation-time analysis a reality [47, 78, 65].

Simulation-time analysis or processing can take place via two modes of operation: in-situ and co-processing [48]. With in-situ processing, code routines are placed directly on the compute nodes and are closely coupled with the simulation execution, thereby allowing in-situ codes to take advantage of massive parallelism. This implies that in-situ codes come with a caveat that each local in-situ process should incur only a negligible run-time overhead as it directly delays the simulation. The other, more flexible method of analysis is co-processing, where the data is moved to a small number of staging resources that are dedicated just for simulation-time processing. These techniques use specially designed high-throughput data transport mechanisms, such as DataTap [3],
DataStager [4], GLEAN [66], or PreData [80] to move the simulation output away to a set of nodes designated for data analysis. As simulations continue with their computation without much delay (provided data processing and I/O times are kept up with the rates of data generation), these staging nodes typically have to perform processing with several orders-of-magnitude less compute power and memory bandwidth. Additionally, not all simulations can easily optimize their codes to employ extra staging resources.

Even though simulation-time analysis involves almost no I/O access, query-driven exploration would still benefit from having the simulation output in a form that is read-efficient across different levels of the memory hierarchy. To perform such transformations without impacting the simulation run-time, placing the processing routines such as indexing, data reduction directly onto the compute nodes, i.e., using *in-situ* processing, can be a viable approach. There is a clear need to have the data “analysis-ready” by the time it leaves the compute cluster, and preparing the data for analysis post-simulation only delays the “time-to-analysis.”

### 1.1.1 *In-situ* Processing for Query-driven Analytics

The primary goal of *in-situ* processing is to reorganize and transform the data to make it more efficient for both real-time and post-simulation time query processing. However, the ideal data layout in memory for simulation computation does not usually translate into an efficient layout on disk for data-intensive analytics. To effectively scale across a large number of processes during computation, simulation scientists employ different levels of domain and data decomposition. The most common decomposition involves dividing the simulation into spatial regions and mapping them to processes. And within each process, arrays of variable values are further sub-divided in memory to optimize for cache and memory accesses. Storing the data as-is from each process may be efficient for writes, but the non-contiguous accesses during reads incur a large number of seeks, thereby resulting in a significant degradation in performance.

Analysis however, especially in the global context, is geared toward extracting few variables across large regions of interests, with additional constraints on variable values. Thus, transforming the data into a more read-efficient organization requires both index generation and efficient aggregation before data is moved out of the compute cluster. An *in-situ* indexing approach has the potential to surmount the index building cost barrier, as it allows the power of the supercomputer cluster running the simulation to be
harnessed, and admits in-memory indexing, eliminating a round-trip to disk.

Designing scalable, *in-situ* algorithms is a challenging task and majority of the current state-of-the-art data management solutions are inapplicable in an *in-situ* or an extreme-scale context because of the following reasons:

1. Increase in compute capabilities largely translates to more processors and cores-per-processor while memory-per-core and processor speeds remain at the same level. Processing data along with simulation execution mandates memory-lightweight and parallelizable solutions, while existing solutions are better suited for the post-processing phase.

2. Even as I/O bandwidth lags behind compute power by several orders of magnitude, existing solutions accelerate query-driven analysis by generating pre-computed indexes that can be significantly larger than the original data. As analysis operations remain highly data-intensive and require efficient out-of-core processing to scale, frequent seeks and large reads over slow storage devices result in increased response times.

3. Before the simulation output leaves the compute cluster, the data must first be made analysis-ready, by aggregating and indexing across spatial, and possibly temporal contexts. Such aggregation schemes must be resource-aware in order to meet network-bandwidth and memory constraints. However, with auxillary indexes that expand the data, efficient *in-network* aggregation and I/O in such scenarios become challenging.

The current state-of-the-art techniques are opposed to one of the central goals of *in-situ* computation: to minimally disturb application run time. Arguably, *a transformative shift is then necessary, making data analytics and data reduction the first class citizens of the data management design and information processing*. In this thesis, we propose query-driven analytics over scientific datasets that is built over large-scale data reduction, performed via *in-situ* computational processing.
1.2 Challenges with Storage-Efficient Scientific Data Analytics

An efficient data management scheme that enables exploratory analytics, driven by indexing and compression, requires careful design of underlying structures for ensuring scalability. In that regard, we review some of state-of-the-art techniques in the literature that specifically relate to scientific data compresion and indexing and we identify their applicability in-situ.

1.2.1 Compression

Using compression to reduce the amount of data accessed from storage can be an effective strategy to address the I/O bottleneck during visualization and analysis. While there are numerous approaches for performing data reduction, we focus primarily on the following techniques that are currently in use with scientific datasets: sampling, lossless compression and lossy compression.

Random sampling is arguably the simplest of all reduction methods. It provides a viable solution for reducing the data size, while preserving some statistical properties of the data (e.g., mean, standard deviation, or density function). However, it is hardly practical for a number of reasons. On the one hand, random data sampling requires random I/O disk access that easily becomes prohibitively expensive for large-scale data sets; in fact, full sequential data scan could provide a faster, yet not practical, solution. On the other hand, features that are of interest to scientists, such as unusual, extreme or rare events, will be likely missed if only a random data sample is being used.

Lossless compression enables applications to save data in a reduced form, but without losing any information or precision. However, the high-entropic nature of spatio-temporal floating-point values results in datasets that can be regarded as “incompressible” by generic lossless compression libraries. While compression techniques specifically designed for scientific datasets have emerged [51, 44, 56], on such hard-to-compress datasets, these techniques reduce data by only around 10-25% of its original size. Note that achieving a 25% reduction in storage and translating it into proportional savings in I/O write time can be significant for large-scale applications that have to periodically checkpoint data. These write-optimized lossless compression techniques are ideally suited for checkpoint and restart datasets that are almost always overwritten and hardly read. But analysis
datasets are written once and read many times, and hence requires larger compression ratios and fast decompression for exploratory querying.

Lossy compression techniques using Discrete Cosine Transform [5], Discrete Wavelet Transform [21], or B-splines [10] discard information content for significantly higher compression ratios compared to lossless compression techniques. While these techniques have been proven in the field of signal processing, they offer poor approximation accuracy on many of the scientific datasets used in this thesis, as these datasets do not emanate from signal sources. This loss in accuracy could lead to interesting phenomena being missed during the analysis phase. Thus, solutions that guarantee error-bounds on every point in the approximated data become necessary to improve quality of exploratory query results, without sacrificing encoding throughput or compression ratios.

### 1.2.2 Indexing

Much of the pioneering work in scientific data indexing comes from the database community even as traditional relational databases are not suitable for scientific datasets for a number of reasons (support for arrays, column based representation for analysis, cost, scalability etc.) [60]. However, one specific form of indexing used in databases, bitmap indexes, introduced in 1985 [62], have shown great promise and wide applicability. Bitmap indexes were designed to target read- or append-only attributes (variables), which aligns with the write-once read-only characteristic of most scientific simulation datasets. These bitmap indexes originally employed no compression and were shown to be effective for low cardinality attributes, i.e., the number of unique values in the variable is only a few. However, with the addition of compression, they were shown to be optimal even for high cardinality attributes [71] and have been effective in accelerating analytical processing on scientific datasets in several cases [74, 50, 52, 70].

To reduce the storage requirements of bitmap indexes on scientific datasets, past studies have focused on three main strategies: *binning*, *encoding*, and *compression* [63].

- **Binning** associates a range of values to a single bin and reduces the number of bitmap vectors that need to be created. This ensures a drastic reduction in storage, especially for floating-point datasets, where the number of unique values can be millions (due to randomness in the mantissa bits). It also helps that queries given by scientists typically do not include more than a few digits of precision, and by
binning based on precision, queries can be answered approximately and quickly by using just the index. The bitmap vector for a bin for a variable consisting of \( N \) elements is then a 0/1 \( N \)-bit vector, with a 1 in the corresponding element position whose value satisfies the bin condition or 0, otherwise.

- **Encoding** defines the relationship between bins. Each bin can represent a unique value or a non-overlapping range of values such as \( \text{temp} = 10.0 \) (equality-encoding), or an overlapping interval of values such as \( 5.0 \leq \text{temp} < 10.0 \) (interval-encoding), or a cumulative range of values like \( \text{temp} \leq 10.0 \) (range-encoding). Both interval-encoding and range-encoding are ideally suited for range queries over floating-point data in terms of computational overhead, but equality encoding tends to take the lowest space requirement amongst the three and offers a good balance between space usage and query performance.

- **Compression** is applied on each bitmap vector by identifying runs, which are consecutive sequences of 0’s or 1’s, and replacing the runs with a run count and a bit value. The state-of-the-art compression with bitmap indexes, the word-aligned hybrid compression (WAH) [71], takes advantage of the fact that memory accesses happen in units of words and aligns literals (sequences containing both 0’s and 1’s) and fills (compressed long runs) to word boundaries. This reduces inefficient memory accesses when operating over compressed bitmaps, and WAH is extremely efficient in performing logical operations directly in the compressed space without having to decode the bitmaps. However, enforcing this alignment improves CPU processing times during querying, increases the storage costs of bitmap indexes and hence poses a challenge in severely I/O bottlenecked environments.

To alleviate the overhead of I/O accesses with WAH-based indexes, several studies have proposed storage optimizations by reducing the number of bitmaps created [59] or accessed during query processing [61]. Encoding forms such as multi-resolution [61] and multi-component bitmap indexes [73] are effective in reducing the amounts of reads by creating bitmaps at different resolutions, but they come at the cost of increased total storage overhead. When the query workloads are known beforehand, bin boundaries can be carefully chosen to reduce storage sizes [59]. This also reduces the number of candidate checks as bin boundaries are chosen to overlap with known query ranges and only a few extra checks of raw data are required to return the result. However, with exploratory
analysis the query ranges and workload cannot possibly be known \textit{a priori}. A few other techniques such as data reorganization to eliminate seeks during candidate checks have also shown promise [75], but they again incur increased storage requirements.

Even with all the above improvements, the storage and encoding overhead of bitmaps makes them inapplicable as an \textit{in-situ} indexing technique.

1.3 Hypothesis

Based on the challenges explained above, the central hypothesis of this thesis can be stated as follows:

Storage-efficient indexing solutions are essential for scalable indexing and query processing for large-scale scientific simulations running on high-performance computing environments. And, utilizing in-situ processing capabilities for data compression and creation of storage-lightweight indexes can significantly reduce both the time-to-analysis and response times for query-driven analytics.

1.4 Proposed Approaches

This thesis contributes new approaches that cover the complementary areas of (a) \textit{in-situ} lossy data compression, (b) query-driven analytics over lossy-compressed data, and (c) parallel indexing using \textit{in-situ} processing. The main components of this thesis (Figure 1.2) and the summary of their approaches and results are described below.

1.4.1 ISABELA for Effective \textit{In-situ} Compression of Scientific Data

We first introduce an effective lossy compression method named ISABELA (\textit{In-situ} Sort-And-B-spline Error-bounded Lossy Abatement) of scientific data [54]. The intuition behind ISABELA stems from the following three observations. First, while being almost random and noisy in its natural form—when sorted—scientific data exhibits a very strong signal-to-noise ratio due to its monotonic and smooth behavior. Second, prior work in curve fitting [29, 69] has shown that monotone curve fitting, such as monotone B-splines, can offer some attractive features for data reduction including, but not limited
to, their goodness of fit with significantly fewer coefficients to store. Finally, the monotonicity property of the sorted data gets preserved in most of its positions with respect to adjacent time steps in many instances. Hence, this property of monotonic inheritance across temporal resolution offers yet another venue for improvement of the overall data compression ratio.

While intuitively simple, ISABELA has addressed a number of technical challenges imposed by end-user requirements. One of the most important factors for the user’s adoption of any lossy data reduction technology is the assurance that the user-acceptable error-bounds are respected. Since curve fitting accuracy is often data-dependent, ISABELA must be robust in its approximation. While curve fitting operations are traditionally time consuming, performing the compression in-situ mandates ISABELA to be fast. Finally, while data sorting—as a pre-conditioner for data reduction—is “a blessing,” it is “a curse” at the same time; reordering the data requires keeping track of the new position indices to associate the decoded data with its original ordering. While management of spline coefficients could be viewed as a light-weight task, the heavy-weight index management forces ISABELA to make some non-trivial decisions between the data compression rates and the data accuracy.

On 20 scientific datasets, ISABELA provides excellent approximation and reduction.
Not only does it consistently outperform Wavelet transform technique, but also delivers better performance, in terms of both compression ratio and accuracy. By capturing the relative per point errors and applying error quantization, ISABELA provides over 75% compression on data from XGC, GTS and FLASH simulation applications, while ensuring 99% accuracy on almost all values. Furthermore, several analytical operations such as correlation and query-driven processing benefit from quick approximate solutions that can be obtained by operating over ISABELA-compressed data. The storage-efficient solution over error-bounded compressed data leads to accurate results on analytical operations over XGC and GTS simulation data sets (> 99% correlation at per-point relative error $\epsilon = 0.1\%$) when compared with the original data. The ISABELA-compressed data and its parallel storage framework are thus ideally suited for scientific data analytics and visualization routines.

This work is published in part at the 17th International European Conference on Parallel and Distributed Computing (Euro-PAR 2011) [54] as a distinguished paper and the full version is published in the Journal of Computation and Concurrency: Practice and Experience [42].

1.4.2 ISABELA-QA: Query-driven Analytics over ISABELA-compressed Data

We propose a parallel query processing engine, called ISABELA-QA that is designed and optimized for knowledge priors driven analytical processing of spatio-temporal, multivariate scientific data that is initially compressed, in-situ by our ISABELA technology. With ISABELA-QA, the total data storage requirement is less than 30% of the original data, which is the eight-fold less than what the existing state-of-the-art data management technologies that require storing both the original data and the index, such as FastBit could offer. Since ISABELA-QA operates on the metadata generated by our compression technology, its underlying indexing technology for efficient query processing is light-weight; namely it requires less than 3% of the original data, unlike existing database indexing approaches that require over 200% of the original data. Moreover, ISABELA-QA is specifically optimized to retrieve the actual values rather than spatial regions for the variables that satisfy user-specified range queries—a functionality that is critical for high-accuracy data analytics. This technology enables query-driven analytics over the compressed spatio-temporal floating-point double- or single-precision data, while offer-
ing a light-weight memory and disk storage footprint solution with parallel, scalable, multi-node, multi-core, and GPU-based query processing.

This work has been published in the Conference on High Performance Computing Networking, Storage and Analysis (SuperComputing) in 2011 [41].

1.4.3 DIRAQ: Scalable Data Encoding for Analytical Query Processing

With ISABELA and ISABELA-QA, we created an inherently lossy compression technique. Based on the lessons learnt, we proposed DIRAQ [40], a parallel, scalable, in-situ index building algorithm, which creates losslessly compressed data with an inbuilt precision-based index for approximate query processing. DIRAQ aggregates group-level indexes across large spatial contexts without significantly impacting the simulation performance. It proceeds by first dividing the processes in the simulation into processor set (pset) groups based on network topology, and then applies in-situ indexing on each local process. The encoding technique converts raw floating-point data into a compressed representation, which incorporates a compressed inverted index to enable optimized range-query access, while also exhibiting a total storage footprint less than that of the original data. Once the indexes are built, the layout of the group-level “defragmented” index layout is created at the group-leader by communicating and merging local index layouts. A load-balanced data transfer mechanism takes place using in-network Remote Memory Access (RMA) operations, to move the index to writer nodes, which then writes the aggregated index to disk. Additionally, we introduced a new approach for aggregator selection that incorporates data-, topology- and memory-awareness to enable smarter aggregation strategies at run-time.

Our proposed method showed promising results on 9 datasets from the FLASH astrophysics simulation [7] and 4 datasets from S3D combustion simulation [34]. Our encoding reduced the overall storage footprint versus the raw data by a factor of 1.1 - 1.8x, and versus indexes created using FastQuery [17] (parallel version of FastBit) by 3 - 6x. Our scalable reorganization and aggregation method combined with our encoding allows up to 6x to-disk throughput improvement compared to MPI-IO on the raw data. Finally, query performance on our defragmented indexes was improved by up to 10x versus FastBit-generated bitmap indexes.

The paper on DIRAQ has received the Best Paper Award at the 22nd Interna-
tional Symposium on High-Performance and Distributed Computing Conference (HPDC 2013) [40].
Chapter 2

ISABELA for Effective In-situ Compression of Scientific Data

2.1 Introduction

Spatio-temporal data produced by large-scale scientific simulations easily reaches terabytes per run. Such data volume poses an I/O bottleneck—both while writing the data into the storage system during simulation and while reading the data back during analysis and visualization. To alleviate this bottleneck, scientists have to resort to subsampling, such as capturing the data every $s^{th}$ timestep. This process leads to an inherently lossy data reduction.

In-situ data processing—or processing the data in-tandem with the simulation by utilizing either the same compute nodes or the staging nodes—is emerging as a promising approach to address the I/O bottleneck [46]. To complement existing approaches, we introduce an effective method for In-situ Sort-And-B-spline Error-bounded Lossy Abatement (ISABELA) of scientific data [54]. ISABELA is particularly designed for compressing spatio-temporal scientific data that is characterized as being inherently noisy and random-like, and thus commonly believed to be incompressible [68]. In fact, the majority of the lossless compression techniques [11, 51, 56] are not only computationally intensive and therefore hardly suitable for in-situ processing, but also are only unable to reduce such data by around 10-25% of its original size (see Section 2.3).

The intuition behind ISABELA stems from the following three observations. First, while being almost random and noisy in its natural form—when sorted—scientific data
exhibits a very strong signal-to-noise ratio due to its monotonic and smooth behavior in its sorted form. Second, prior work done in curve fitting [29, 69] has shown that monotone curve fitting, such as monotone B-splines, can offer some attractive features for data reduction including, but not limited to, their goodness of fit with significantly fewer coefficients to store. Finally, the monotonicity property of the sorted data gets preserved in most of its positions with respect to adjacent time steps in many instances. Hence, this property of monotonic inheritance across temporal resolution offers yet another venue for improvement of the overall data compression ratio.

While intuitively simple, ISABELA has addressed a number of technical challenges imposed by end-user requirements. One of the most important factors for the user’s adoption of any lossy data reduction technology is the assurance that the user-acceptable error-bounds are respected. Since curve fitting accuracy is often data-dependent, ISABELA must be robust in its approximation. While curve fitting operations are traditionally time consuming, performing the compression in-situ mandates ISABELA to be fast. Finally, while data sorting—as a pre-conditioner for data reduction—is “a blessing,” it is “a curse” at the same time; reordering the data requires keeping track of the new position indices to associate the decoded data with its original ordering. While management of spline coefficients could be viewed as a light-weight task, the heavy-weight index management forces ISABELA to make some non-trivial decisions between the data compression rates and the data accuracy.

2.2 A Motivating Example

Much of the work for in-situ data reduction in this study stems from particle simulation codes, specifically Gyrokinetic Tokamak Simulation (GTS) [67] and XGC1 [53] which respectively simulate micro-turbulence of magnetically confined fusion plasmas of toroidal devices in cores and edges of fusion reactors. Production runs of these simulation applications typically consume hundreds of thousands of cores on petaflop systems, such as NCCS/ORNL Jaguar [9], utilizing the ADIOS library [45] for performing I/O intensive operations.

Simulation data sets can be broadly divided into: (1) checkpoint data to restart the simulation in case of an execution failure (C&R); (2) analysis (A) data, such as density and potential fluctuations, for performing various post-processing physics analyses and
(3) diagnostics data used, for example, for code validation and verification (V&V) (see Table 2.1).

Table 2.1: Summary of GTS output data by different categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Write Frequency</th>
<th>Read Access</th>
<th>Size/Write</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;R</td>
<td>Every 1-2 hours</td>
<td>Once or never</td>
<td>A few TBs</td>
<td>≈TBs</td>
</tr>
<tr>
<td>A</td>
<td>Every 10th time step</td>
<td>Many times</td>
<td>A few GBs</td>
<td>≈TBs</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Every 2nd time step</td>
<td>A few times</td>
<td>A few MBs</td>
<td>≈GBs</td>
</tr>
</tbody>
</table>

Unlike C&R data that requires lossless compression, analysis (A) data is inherently lossy, and as such, it can tolerate some error-bounded loss in its accuracy. What is more important is that it is the analysis data that is being accessed many times by different scientists using various analysis and visualization tools or Matlab physics analysis codes. Therefore, aggressive data compression that could enable interactive analytical data exploration is of paramount concern, and is therefore the main focus of ISABELA. For illustrative purposes, in the rest of the paper we will use linearized 64-bit double precision floating point arrays. The temporal snapshots consist of series of 172,111 values organized one-dimensionally of both Potential and Density fluctuations from GTS analysis data, a series of 4,096 values organized three-dimensionally from Flash astrophysics data, and a series of 124,701 turbulence intensity values organized one-dimensionally from XGC1 analysis data.

2.3 Problem Statement

The inherent complexity of scientific spatio-temporal data drastically limits the applicability of both lossless and lossy data compression techniques and presents a number of challenges for new method development. Such data not only consists of floating-point values, but also exhibits randomness without any distinct repetitive bit and/or byte patterns (also known as high entropy data, and hence, uncompressible [19, 55]). Thus, applying standard lossless compression methods does not result in an appreciable data reduction.

Table 2.2 illustrates the compression rates achieved and the time required to compress and decode 12,836KB of GTS analysis data by state-of-the-art methods. In addition, sci-
entific data often exhibits a large degree of fluctuations in values across even directly adjacent locations in the array. These fluctuations render lossy multi-resolution compression approaches like Wavelets [24] ineffective.

The compression ratio $CR_M(D)$ of a compression method $M$ for data $D$ of size $|D|$ reduced to size $|D_M|$ is defined by Equation 2.1:

$$CR_M(D) = \left| \frac{|D| - |D_M|}{|D|} \right| \times 100\%.$$ (2.1)

The accuracy of lossy encoding techniques is measured using Pearson’s correlation coefficient ($\rho$) and Normalized Root Mean Square Error between an $N$-dimensional original data vector $D = (d_0, d_1, \ldots, d_{N-1})$ and decompressed data vector $D' = (d'_0, d'_1, \ldots, d'_{N-1})$ defined by Equations 2.2, and 2.3:

$$\rho(D) = \frac{\text{covariance}(D, D')}{\text{std. dev}(D) \text{std. dev}(D')} = \frac{1}{N} \sum_{i=0}^{N-1} (d_i - \bar{D})(d'_i - \bar{D'})}{\sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (d_i - \bar{D})^2} \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (d'_i - \bar{D'})^2}},$$ (2.2)

where $D = \frac{\sum_{i=0}^{N-1} d_i}{N}$, and $D' = \frac{\sum_{i=0}^{N-1} d'_i}{N}$

$$NRMSE_M(D) = \frac{\text{RMSE}_M(D, D')}{\text{Range}(D)} = \frac{\sqrt{\sum_{i=0}^{N-1} (d_i - d'_i)^2}}{\max(D) - \min(D)}$$ (2.3)

Capturing both $NRMSE$ and $\rho$ provides not only the extent of error but also measures the degree to which the original and approximated data are linearly related. Thus, achieving $NRMSE_M(D) \sim 0$, $\rho(D) \sim 1$ and $CR \sim 100\%$ would indicate ideal performance.

Table 2.2: Performance of examplar lossless and lossy data compression methods.

<table>
<thead>
<tr>
<th>Metric</th>
<th>FPC</th>
<th>LZMA</th>
<th>ZIP</th>
<th>BZ2</th>
<th>ISABELA</th>
<th>Wavelets</th>
<th>B–splines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lossless?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$CR_M$ (%)</td>
<td>3.12</td>
<td>2.72</td>
<td>1.13</td>
<td>1.11</td>
<td>81.44*</td>
<td>22.51*</td>
<td>0*</td>
</tr>
<tr>
<td>$T_C$ (sec.)</td>
<td>0.58</td>
<td>7.01</td>
<td>1.03</td>
<td>3.96</td>
<td>0.93</td>
<td>0.62</td>
<td>0.78</td>
</tr>
<tr>
<td>$T_D$ (sec.)</td>
<td>0.56</td>
<td>1.38</td>
<td>0.49</td>
<td>1.18</td>
<td>1.05</td>
<td>0.58</td>
<td>0.82</td>
</tr>
</tbody>
</table>

$CR_M$ = compression ratio of method $M$, $T_C$ = compression time, $T_D$ = decompression time.

*CR achieved by lossy models for 0.99 correlation and 0.01 NRMSE fixed accuracy. All runs are performed on an Intel Core 2 Duo 2.2 GHz processor with 4 GB RAM, openSUSE Linux v11.3.
2.4 Theory & Methodology

Existing multi-resolution compression methods often work well on image data or time-varying signal data. For scientific data-intensive simulations, however, data compression across the temporal resolution requires data for many timesteps be buffered in memory, which is, obviously, not a viable option. Applying lossy compression techniques on this data across the spatial resolution requires a significant tradeoff between the compression ratio and the accuracy. Hence, to extract the best results out of the existing approximation techniques, a transformation of this data layout becomes necessary.

Figure 2.1: A slice of GTS Potential: (A) original; (B) sorted; (C) decoded after B–splines fitting to original; and (D) decoded after B–splines fitting to sorted.

2.4.1 Sorting-based Data Transformation

Sorting changes the data distribution in the spatial domain, from a highly irregular signal (Figure 2.1, A) to a smooth and monotonous curve (Figure 2.1, B). The rationale behind sorting—as a pre-conditioner for a compression method—is that fitting on a monotonic
curve can provide a model that is more accurate than one on unordered and randomly distributed data. Figure 2.1 illustrates the significant contrast in how closely \((D)\) or poorly \((C)\) the decompressed data approximates the original data when the \(B\)-splines curve fitting [10] operates on sorted versus unsorted data, respectively.

### 2.4.2 Background: Cubic \(B\)-splines Fitting

Sorting the data in an increasing order provides a sequence of values whose rate of change is guaranteed to be the slowest. Although this sequence resembles a smooth curve, performing curve fitting using non-linear polynomial interpolation becomes difficult for complex shape curves. Computing interpolation constants for higher-order polynomials in order to fit these complex curves is computationally intensive for \textit{in-situ} processing.

A more effective technique is spline curve fitting. A \textit{spline curve} is a sequence of curve segments joined together via \textit{knots} that produce a single continuous curve. Each piece of the curve segment is then defined via a lower-order polynomial function. Using several lower-order polynomial functions to fit smaller regions of the curve tends to perform highly efficiently when compared to using a higher-order polynomial function to fit the entire curve.

A \(B\)-spline curve is a sequence of piecewise parametric curves. A \textit{cubic} \(B\)-spline is composed of polynomial functions of degree \(d = 3\), which have faster interpolation time and produce “smooth” curves (i.e., second-order differentiable) at the knot locations. Knot locations are points in the parameter space that describe the start and end of a curve segment.

As an example, consider the cubic \(B\)-spline curve in Figure 2.2 with six control points, \(P_1, \ldots, P_6\), and ten knots, \(u_1, \ldots, u_{10}\), \(u_1 = \cdots = u_4 = 0\) and \(u_7 = \ldots = u_{10} = 1.0\). The knot points are the end-points of the piecewise curve segments, \(S_1, S_2,\) and \(S_3\), given by \(S_1 = s_1s_2, S_2 = s_2s_3,\) and \(S_3 = s_3s_4\), each defined in its parameter space, \(u \in [0, 1]\). The control points \(P\) define the shape of the curve.

More formally, given a set of \(m\) control points \(P = \{P_1, P_2, \ldots, P_m\}\), and a knot vector \(U\) with its sequence of knots, \(u_1 \leq u_2 \leq \ldots \leq u_k, k = m + d + 1, \forall u_i \in [0, 1]\), the \(B\)-spline parametric curve, \(S : [u_1, u_k] \rightarrow \mathbb{R}^2\), is defined by a linear combination of its
basis functions $B_{i,j}(u)$ via Equation 2.4:

$$S(u) = \sum_{i=1}^{m} B_{i,d}(u)P_i, \text{ where}$$

$$B_{i,0}(u) = \begin{cases} 1, & \text{if } u_i \leq u < u_i+1 \\ 0, & \text{otherwise} \end{cases}$$

$$B_{i,j}(u) = \frac{u - u_i}{u_i + j - u_i}B_{i,j-1}(u) + \frac{u_{i+j+1} - u}{u_{i+j+1} - u_{i+1}}B_{i+1,j-1}(u),$$

where $j = \overline{1, d}$ and $i = \overline{1, m}$, and the basis functions $B_{i,j}(u)$ determine the extent to which the control points $P$ affect the shape of the curve. Thus, splines can control the local shape of the curve without affecting the shape of the curve globally. This also implies that both curve fitting and interpolation are efficient operations and can provide location-specific data decoding without decompressing all the data. While sorting does rearrange the points, location-specific decoding is still possible by performing an additional single level translation of data location from the original to the sorted vector, and then retrieving the interpolated value on the sorted $B$-spline curve.

Figure 2.2: A cubic $B$-spline fitting with $m = 6$ control points, $k = 10$ knots, and with 3 piecewise cubic segments.
2.4.3 Maximizing Compression Ratio via Window Splitting

In this section, we look at approaches for maximizing the compression ratio, while maintaining an accurate approximation model. Let us assume that the original data \( D \) is a vector of size \( N \), namely \( D = (d_0, d_1, \ldots, d_{N-1}) \). This way, we can associate a value \( d_i \) with each index value \( i \in I = \{0, 1, \ldots, N - 1\} \). Let us also assume that each vector element, \( d_i \in \mathbb{R} \), is stored as a 64-bit double-precision value. Therefore, storing the original data requires \(|D| = N \times 64\) bits.

Assuming that \( D \) is a discrete approximation of some curve, its \( B \)-splines interpolation \( D_B \) requires storing only \( B \)-splines constants—the knot vector and the basis coefficients—in order to reconstruct the curve. Let \( C \) denote the number of such 64-bit double-precision constants. Then, storing the compressed data after \( B \)-splines curve fitting requires \(|D_B| = C \times 64\) bits.

The random-like nature of \( D \) (see Figure 2.1, (A)) requires \( C \sim N \) to provide accurate lossy compression, and hence, leads to a poor compression rate (see Table 2.2, last column). However, due to the nature of scientific datasets, applying \( B \)-splines interpolation after sorting \( D \) requires only a few constants, \( C = O(1) \ll N \), in order to provide high decompression accuracy (see Figure 2.1, (D)).

While significantly reducing the number of \( B \)-splines constants \( C \), sorting \( D \) will reorder the vector elements via some permutation \( \pi \) of its indices, namely \( I \xrightarrow{\pi} I_\pi = \{i_1, i_2, \ldots, i_N\} \), such that \( d_{i_j} \leq d_{i_{j+1}}, \forall i_j \in I_\pi \). As a result, we need to keep track of the new index \( I_\pi \) so that we could associate the decompressed sorted vector \( D_\pi \) back to the original vector \( D \) by using its correct index \( I \). Since each index value \( i_j \) requires \( \log_2 N \) bits, the total storage requirement for \( I_\pi \) is \(|I_\pi| = N \times \log_2 N \) bits. Therefore, the vector length \( N \) is the only factor that determines the storage requirements for the index \( I_\pi \).

One way to optimize the overall compression ratio, \( CR_{ISABELA} \), is to first split the entire vector \( D \) into fixed-sized windows of size \( W \) (rounding up the size of the tail window for the simplicity of analysis), or \( D = \bigcup D^k, D^i \cap D^j = \emptyset, I^k = \{(k-1)W_0, (k-1)W_0 + 1, \ldots, kW_0 - 1\}, i, j, k \in \mathbb{N}_W, i \neq j, \text{ and } N_W = \left\lceil \frac{N}{W_0} \right\rceil \). Then, the \( B \)-splines interpolation is applied to each window \( D^k \) separately.

With this strategy, ISABELA’s storage requirement for the compressed data is de-
fined by Equation 2.6:

\[ |D_{\text{ISABELA}}| = \sum_{k=1}^{N_W} (|D_B^k| + |I_{\pi}^k|), \quad (2.6) \]

\[ = N_W \times (C \times 64 + W_0 \times \log_2 W_0) \]

Substituting Equation 2.6 into Equation 2.1 and simplifying the resulting equation, we obtain the following compression ratio for ISABELA defined by Equation 2.7:

\[ CR_{\text{ISABELA}}(D) = (1 - \frac{\log_2(W_0)}{64} - \frac{C}{W_0}) \times 100\% \quad (2.7) \]

From Equation 2.7, we can analytically deduce the trade-off between the window size \( W_0 \) and the number of \( B \)-splines constants \( C \) that give the best compression ratio. For example, for \( W_0 > 65,536 \), the size of the index alone would consume more than 25% of the original data. By fixing \( W_0 = 1024 \), we balance the computational cost of sorting windows and the storage cost for recording the index and the fitting coefficients, which results in an overall compression rate of 81.4% per time step. And empirically, we found that \( C = 30 \) and \( W_0 = 1024 \) allows ISABELA to achieve both \( > 0.99 \) correlation and \( < 0.05 \) NRMSE between the original and decompressed GTS data, thus balancing for both accuracy and storage.

### 2.4.4 Error Quantization for Guaranteed Point-by-point Accuracy

The above sorting-based curve fitting model ensures accurate approximation only on a per window basis and not on a per point basis. As a result, in certain locations, the \( B \)-splines estimated data deviates from the actual by a margin exceeding a defined tolerance. For example, almost 95% of the approximated GTS Potential values average a 2% relative error, where the percentage of the relative error (\( \epsilon \)) at each index \( i \) between \( D = (d_0, d_1, ..., d_{N-1}) \) and \( D_{\text{ISABELA}} = (d'_0, d'_1, ..., d'_{N-1}) \) is defined as \( \epsilon_i = \frac{d_i - d'_i}{d_i} \times 100\% \). While the number of such location points is reasonably low due to accurate fitting achieved by \( B \)-splines on monotonic data, ISABELA guarantees that a user-specified point-by-point error is respected by utilizing an error quantization strategy.

Storing relative errors between estimated and actual values enables us to reconstruct
the data with high accuracy. Quantization of these errors into 32-bit integers results in a large degree of repetition, where majority of the values lie between [-2, 2]. These integer values lend themselves to high compression rates (75% − 90%) with standard lossless compression libraries. These compressed relative errors are stored along with the index during encoding. Upon decoding, applying these relative errors ensures decompressed values to be within a user-defined threshold $\tau_e$ for per point relative error.

Parallel Compression

Fixing the window size, and the number of coefficients in a window keeps the compression design embarrassingly parallel where each window can be compressed independently. However, with the inclusion of error quantization, the size of the encoded errors and hence the size of the compressed window, no longer remains a constant. To write the compressed data into a contiguous location each thread must know a priori as to where to start writing the $B$–spline coefficients, index mapping and the encoded errors. This incurs communication overhead when parallelizing the operation on a per-window level. To overcome the delay due to communication, each thread is assigned the task of compressing a fixed number of windows and writing the compressed data including the encoded errors to a local memory space. This data is then written out to disk or passed to I/O libraries by iterating through the local memory of each thread. The number of available threads is usually limited when compared to the number of windows of data; therefore, iterating on a per-thread level incurs less overhead than moving the data around for contiguous I/O. In fact, placing compression routines on nodes which produce data has been shown to be effective in reducing the time taken to move data onto the disk [57]. Given that ISABELA achieves a higher degree of data reduction than the ISOBAR [56] lossless compression technique proposed by Schendel et al., further improvements in writing time can be expected.

2.4.5 Exploiting $\Delta$–encoding for Temporal Index Compression

To a large extent, the ordering of the sorted data values is similar between adjacent timesteps, i.e., the monotonicity property of the data extends to index integer values. Hence, we apply a differential encoding scheme to the index vector $I_x$ before compressing the index using standard lossless compression libraries. [Note that subsequent scheme is applied to each individual data window $D^n$.]
Suppose that at timestep $t_0$, we first build the index $I_\pi(t_0)$ consisting of no redundant values, which is essentially, incompressible. Hence, this index is stored as is. However, at the next timestep $t_0 + 1$, the difference in index values $\Delta I_{+1} = I_\pi(t_0 + 1) - I_\pi(t_0)$ is small (see Figure 2.3) due to monotonicity of the original data values $D^n$ and, hence, the sorted values across adjacent timesteps.

Thus, instead of storing the index values at each timestep, we store the index values at $t_0$, denoted as the reference index, along with the compressed pairwise index differences $\Delta I_{+1}$ between adjacent timesteps. But, in order to recover the data at time $t_0 + \delta t$, we must read in both the reference index $I_\pi(t_0)$ and all the first-order differences $\Delta I_{+1}$ between adjacent timesteps in the time window $(t_0, t_0 + \delta t)$. Therefore, the higher value of $\delta t$ will adversely affect reading time. To address this problem, we instead store and compress a higher-order difference, $\Delta I_{+j} = I_\pi(t_0 + j) - I_\pi(t_0)$, where $j \in (1, \delta t)$, for the growing value of $\delta t$ until the size of the compressed index crosses a user-defined threshold. Once the threshold is crossed, the index for the current timestep is stored as is, and is considered as the new reference index.

### 2.4.6 ISABELA Data Workflow

Figure 2.4 depicts the overall data workflow behind the ISABELA compression engine, starting from data generation to the organization of compressed data in storage. ISABELA compression is characterized by a communication-free model which can be easily parallelized using OpenMP, GPU and multi-node configurations. The parallelism
can be extended to the I/O layer as well, thus enabling efficient data analysis. The subsequent section discusses some of the design considerations for each component of the workflow, along with their impacts on storage and accuracy. as is and is made the new

![Workflow Diagram]

Figure 2.4: Workflow of ISABELA compression engine from data generation to *in-situ* compression to storage.

reference index.

### 2.5 Results

Evaluation of a lossy compression algorithm primarily depends on the accuracy of the fitting model and the compression ratio ($CR$) achieved. As this compression is performed *in-situ*, analysis of the time taken to perform the compression assumes significance as well. Here, we evaluate ISABELA with emphasis on the aforementioned factors, using normalized root mean standard error ($NRMSE$) and Pearson correlation ($\rho$) between the original and decompressed data as accuracy metrics.
2.5.1 Per Window Accuracy

In this section, we compare the Pearson correlation (\(\rho\)) between the original and decompressed data using Wavelets and B–splines on original data and using ISABELA. The following parameters are fixed in this experiment: \(W_0 = 1024\), \(C_{B\text{-}spline} = 150\), and \(C_{ISABELA} = 30\). This fixes \(CR = 81\%\), and wavelet coefficients values are thresholded to achieve the same compression rate as well. Figure 2.5a illustrates that ISABELA performs exceptionally well even for much smaller \(C\) values due to the monotonic nature of the sorted data. In fact, \(\rho\) is > 0.99 for almost all the windows. However, both Wavelets and B–splines exhibit a large degree of variation and poor accuracy across different windows. This translates to NRMSE values that are one-to-two orders of magnitude larger than the average 0.005 NRMSE value produced by ISABELA.

ISABELA performs exceptionally well on data from the linear stages of the GTS simulation (first few thousand timesteps), as shown in Figure 2.5b. Yet, the performance for the non-linear stages (timestep \(\approx 10,000\)), where the simulation is characterized by a large degree of turbulence is of particular importance to scientists. Figure 2.5b, with intentionally magnified correlation values, shows that accuracy for the non-linear stages across windows drops indeed. Unlike Wavelets (Figure 2.5a), however, this correlation does not drop below 0.92.
2.5.2 Effect of Window Size $W_0$ on Accuracy

In order to keep the compression design embarrassingly parallel, the size of the window $W_0$ is kept fixed. The choice of window size should not only have a low index storage footprint, but must also contain sufficient number of points to approximate the curve accurately. In Figure 2.6, we calculate the $NRMSE$ sensitivity to different values of constants $C = 20, 30, 2 \times \log_2(W_0)$ and to different window sizes $W_0$ with ISABELA.

One would expect that increasing $W_0$ without increasing $C$ would increase $NRMSE$. But this is not the case. As it turns out, the data becomes highly smooth with larger $W_0$. Hence, $NRMSE$ initially increases, but then it levels off as $W_0$ keeps growing. When $C = 2 \times \log_2(W_0)$, that is the number of constants is kept constant proportional to the window size, $NRMSE$ decreases as $W$ grows. The choice of $W = 1024$ and $C = 30$ balances the trade-off between accuracy and compression rate, providing a fixed 81% reduction in the size of the data.

![Figure 2.6: Sensitivity of NRMSE values for ISABELA-compressed GTS Potential data across 100 windows over varying window sizes. The compression rate is fixed at CR = 81.44%](image)

2.5.3 Trade-off between Compression and Per Point Accuracy

To alleviate the aforementioned problem, we apply error quantization, as described in Sec. 2.4.4. In both linear and non-linear stages of GTS simulation, the compression ratios...
Figure 2.7: Compression ratio (CR) performance: (a) For various per point relative error thresholds ($\tau_\epsilon$) in GTS Potential during linear and non-linear stages of the simulation. (b) For various timesteps with $\tau_\epsilon = 1\%$ at each point (for GTS Potential: $t_1 = 1,000, \Delta t = 1,500$; for Velocity in Flash: $t_1 = 3,000, \Delta t = 3,500$.

are similar when the per point relative error ($\tau_\epsilon$) is fixed (see Figure 2.7a). This is because the relative error in consecutive locations for the sorted data tends to be similar. This property lends well to encoding schemes. Thus, even when the error tends to be higher in the non-linear stage, compared with the linear stage, the compression rates are highly similar. For $\tau_\epsilon = 0.1\%$ at each point, the CR lowers to an around 67.6%. This implies that by capturing 99.9% of the original values, the data from the simulation is reduced to less than one-third of its total size.

Figure 2.7b shows the compression ratio (with $\tau_\epsilon = 1\%$) over the entire simulation run using the GTS fusion simulation and Flash astrophysics simulation codes. For GTS Potential data, the compression ratio remains almost the same across all stages of the simulation. With Flash, after error quantization, most relative errors are 0’s. Compressing these values results in negligible storage overhead, and hence CR remains at 80% for the majority of timesteps.

2.5.4 Effect of $\Delta$-encoding on Index Compression

In this section, we show that compressing along the time dimension further improves ISABELA’s overall compression of spatio-temporal scientific datasets by up to 2%—5%.
For example, on density with $W_0 = 1024$, the index size reduces from 15.63% to 10.63%–13.63% with index compression. Table 2.3 show the compression rates achieved for different orders of $\Delta I_{+j}, j = 1, 2, 3$. While increasing $W_0$ improves spatial compression to a certain extent, it severely diminishes the reduction of the index along the temporal resolution. This is due to the fact that with larger windows and a larger $\delta t$ between timesteps, the difference in index values lacks the repetitiveness necessary to be compressed well by standard lossless compression libraries.

<table>
<thead>
<tr>
<th>$W_0$</th>
<th>Without $\Delta$–encoding</th>
<th>$\Delta I_{+1}$</th>
<th>$\Delta I_{+2}$</th>
<th>$\Delta I_{+3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>80.08 (80.08)</td>
<td>81.83 (84.14)</td>
<td>81.87 (85.09)</td>
<td>81.68 (85.36)</td>
</tr>
<tr>
<td>1024</td>
<td>81.44 (81.44)</td>
<td>83.14 (85.65)</td>
<td>83.21 (86.57)</td>
<td>82.98 (86.76)</td>
</tr>
<tr>
<td>2,048</td>
<td>81.34 (81.34)</td>
<td>83.03 (85.56)</td>
<td>83.07 (86.44)</td>
<td>82.88 (86.66)</td>
</tr>
<tr>
<td>4,096</td>
<td>80.51 (80.51)</td>
<td>82.14 (84.64)</td>
<td>82.21 (85.51)</td>
<td>82.03 (85.76)</td>
</tr>
<tr>
<td>8,192</td>
<td>79.32 (79.32)</td>
<td>80.99 (83.38)</td>
<td>81.04 (84.24)</td>
<td>80.83 (84.46)</td>
</tr>
</tbody>
</table>

### 2.5.5 Compression Time

The overhead induced on the runtime of the simulation due to *in-situ* data compression is the net sum of the times taken to sort $D$, build $I_\pi$ and perform cubic $B$–spline fitting. However, for a fixed window size $W_0$, sorting and building the index is computationally less expensive compared to $B$–spline fitting. When executed in serial, ISABELA compresses data at average of around 12 MB/s rate, the same as gzip compression level 6, as shown in Table 2.2. Within the context of the running simulation, if each core generates around 10 MB of data every 10 seconds, it can be reduced to $\approx 2$ MB in 0.83 seconds using ISABELA. Additionally, to further reduce the impact of *in-situ* compression on the main computation, ISABELA can be executed at the I/O nodes rather than at the compute nodes [3, 80].

### 2.5.6 Performance for Fixed Compression

In this section, we evaluate the performance of ISABELA and Wavelets on 13 public scientific datasets (from numerical simulations, observations, and parallel messages) [11],
and 7 datasets from petascale simulation applications for the fixed \( CR = 81\% \). For the Wavelet transform, we use the “fields” library package in R. To fix the compression ratio, the wavelet coefficients with the lowest absolute values are reduced to 0. We then compare the averages of \( \rho_a \) and \( NRMSE_a \) of ISABELA and Wavelets across 400 windows (see Table 2.4). Out of the 20 datasets, eighteen (three) datasets exhibit \( \rho_a = 0.98 \) with ISABELA (Wavelets). The \( NRMSE_a \) values for Wavelets are consistently an order of magnitude higher than for ISABELA. Wavelets outperform ISABELA on obs_spitzer, which consists of a large number of piecewise linear segments for most of its windows. Cubic \( B-splines \) do not estimate well when data segments are linear.

Table 2.4: ISABELA vs. Wavelets for fixed \( CR = 81\% \) and \( W_0 = 1,024 \).

<table>
<thead>
<tr>
<th></th>
<th>( \rho_a )</th>
<th>( NRMSE_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wavelets</td>
<td>ISABELA</td>
</tr>
<tr>
<td>msg_sppm</td>
<td>0.400 ± 0.287</td>
<td>0.982 ± 0.017</td>
</tr>
<tr>
<td>msg_bt</td>
<td>0.754 ± 0.371</td>
<td>0.981 ± 0.054</td>
</tr>
<tr>
<td>msg_lu</td>
<td>0.079 ± 0.187</td>
<td>0.985 ± 0.031</td>
</tr>
<tr>
<td>msg_sp</td>
<td>0.392 ± 0.440</td>
<td>0.967 ± 0.051</td>
</tr>
<tr>
<td>msg_sweep3d</td>
<td>0.952 ± 0.070</td>
<td>0.998 ± 0.006</td>
</tr>
<tr>
<td>num_brain</td>
<td>0.994 ± 0.008</td>
<td>0.983 ± 0.028</td>
</tr>
<tr>
<td>num_comet</td>
<td>0.988 ± 0.018</td>
<td>0.994 ± 0.025</td>
</tr>
<tr>
<td>num_control</td>
<td>0.614 ± 0.219</td>
<td>0.993 ± 0.017</td>
</tr>
<tr>
<td>num_plasma</td>
<td>0.605 ± 0.062</td>
<td>0.994 ± 0.004</td>
</tr>
<tr>
<td>obs_error</td>
<td>0.278 ± 0.203</td>
<td>0.994 ± 0.004</td>
</tr>
<tr>
<td>obs_info</td>
<td>0.717 ± 0.136</td>
<td>0.993 ± 0.006</td>
</tr>
<tr>
<td>obs_spitzer</td>
<td>0.992 ± 0.001</td>
<td>0.742 ± 0.004</td>
</tr>
<tr>
<td>obs_temp</td>
<td>0.611 ± 0.114</td>
<td>0.994 ± 0.011</td>
</tr>
<tr>
<td>gts_phi</td>
<td>0.886 ± 0.030</td>
<td>0.998 ± 0.003</td>
</tr>
<tr>
<td>gts_zion</td>
<td>0.246 ± 0.024</td>
<td>0.996 ± 0.003</td>
</tr>
<tr>
<td>gts_zeon</td>
<td>0.232 ± 0.039</td>
<td>0.995 ± 0.003</td>
</tr>
<tr>
<td>xgc_iphase</td>
<td>0.235 ± 0.027</td>
<td>0.992 ± 0.013</td>
</tr>
<tr>
<td>flash_gamec</td>
<td>0.918 ± 0.063</td>
<td>0.989 ± 0.010</td>
</tr>
<tr>
<td>flash_velx</td>
<td>0.893 ± 0.055</td>
<td>0.999 ± 0.005</td>
</tr>
<tr>
<td>flash_vely</td>
<td>0.937 ± 0.054</td>
<td>0.993 ± 0.008</td>
</tr>
</tbody>
</table>

30
Figure 2.8: Compression ratio (CR) performance with per-window constraint $\rho > 0.99$, and $NRMSE < 0.01$: (a) On each window, with varying number of coefficients, in Flash velx (b) Overall storage cost over 400 windows across various petascale simulation datasets.

2.5.7 Performance for Fixed Accuracy

From the end-user perspective, the input arguments are defined by accuracy levels. In this section, we evaluate the storage footprint of ISABELA under strict accuracy constraints of $\rho > 0.99$, and $NRMSE < 0.01$. Figure 2.8a contrasts the storage consumed by Wavelets against ISABELA, when the number of coefficients saved per window with both methods are made flexible. In the case of ISABELA, the compression ratio in each window remains close to the fixed values ($C = 30$). Even on hard-to-compress datasets like gts_zion, as seen in Figure 2.8b, ISABELA offers 3x reduction compared to Wavelets at the same accuracy levels.

2.5.8 Scientific Data Analysis on ISABELA-compressed Data

Scientific data is qualitatively analyzed using visualization routines and statistically using scripting languages. More often than not, scientists need to explore the data in order to formulate their hypotheses, which are subsequently analyzed using the above routines. To perform exploratory data analysis, large volumes of scientific data need to be queried efficiently and repeatedly. In this section, we take a detailed look at how compressing
data with ISABELA affects the analysis of scientific data from different simulations.

Figure 2.9: Comparison of intensity plots of the normalized temperature values in XGC data, across temporal and poloidal dimensions for original (left) and ISABELA (middle) decompressed data. The absolute difference between the derived values, on an order of magnitude lower scale, are shown in the figure on the right.

Analysis of XGC-1 Turbulence Datasets

The XGC-1 code is a Particle-In-Cell simulation of neoclassical and electrostatic turbulence physics, designed for tokamak edge plasmas in nuclear reactors. In the case of XGC fusion simulation, we compare the visuals of the variation of quantities derived from temperature across temporal and poloidal dimensions. Figures 2.9 show the plot where the field temperature values are interpolated, and normalized for varying poloidal flux values and time. The left figure represents the visual analysis when performed on the original data set, and the middle figure represents the same visual but on ISABELA-compressed data. The rightmost figure shows the difference in absolute values between the two sets of values, captured on an order of magnitude lower scale than the original values. Clearly, quantities derived from ISABELA-compressed data capture the same physical phenomena as the actual data.

Another output produced by the simulation is the turbulence of particles across grids that are divided into radial zones. To analyze this data, scientists often look at the time-based correlation of normalized turbulence intensity values across surfaces spanning different radial zones. Table 2.5 shows the correlation matrix on average turbulence intensity values between five radial zones. Table 2.6 holds the same correlation matrix, but
on ISABELA-compressed data. The absolute difference of the derived analysis between ISABELA-compressed and original data is minimal.

Table 2.5: XGC data analysis showing the correlation between radial zones on original data

<table>
<thead>
<tr>
<th>zone</th>
<th>zone1</th>
<th>zone2</th>
<th>zone3</th>
<th>zone4</th>
<th>zone5</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone1</td>
<td>1</td>
<td>0.64</td>
<td>0.35</td>
<td>0.60</td>
<td>0.82</td>
</tr>
<tr>
<td>zone2</td>
<td>0.64</td>
<td>1</td>
<td>-0.05</td>
<td>0.38</td>
<td>0.86</td>
</tr>
<tr>
<td>zone3</td>
<td>0.35</td>
<td>-0.05</td>
<td>1</td>
<td>0.62</td>
<td>0.05</td>
</tr>
<tr>
<td>zone4</td>
<td>0.60</td>
<td>0.38</td>
<td>0.62</td>
<td>1</td>
<td>0.54</td>
</tr>
<tr>
<td>zone5</td>
<td>0.82</td>
<td>0.86</td>
<td>0.05</td>
<td>0.54</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.6: Impact of error quantization on correlation between radial zones on ISABELA-compressed data, and the difference with the correlation over original data.

<table>
<thead>
<tr>
<th>zone</th>
<th>zone1</th>
<th>zone2</th>
<th>zone3</th>
<th>zone4</th>
<th>zone5</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone1</td>
<td>1 (0.000)</td>
<td>0.64 (0.053)</td>
<td>0.35 (0.005)</td>
<td>0.60 (0.015)</td>
<td>0.82 (0.074)</td>
</tr>
<tr>
<td>zone2</td>
<td>0.64 (0.053)</td>
<td>1 (0.000)</td>
<td>-0.05 (0.098)</td>
<td>0.38 (0.032)</td>
<td>0.86 (0.071)</td>
</tr>
<tr>
<td>zone3</td>
<td>0.35 (0.005)</td>
<td>-0.05 (0.098)</td>
<td>1 (0.000)</td>
<td>0.62 (0.022)</td>
<td>0.05 (0.005)</td>
</tr>
<tr>
<td>zone4</td>
<td>0.60 (0.015)</td>
<td>0.38 (0.032)</td>
<td>0.62 (0.022)</td>
<td>1 (0.000)</td>
<td>0.54 (0.079)</td>
</tr>
<tr>
<td>zone5</td>
<td>0.82 (0.074)</td>
<td>0.86 (0.071)</td>
<td>0.05 (0.005)</td>
<td>0.54 (0.079)</td>
<td>1 (0.000)</td>
</tr>
</tbody>
</table>

We use another example of the XGC data analysis routine, similar to the scenario mentioned above. Apart from calculating the correlation, the normalized turbulence intensity values across different regions of the grid are visualized as a time series. Figure 2.10 compares the plot of turbulence intensity values over 620 timesteps for original, and ISABELA-compressed data with $\epsilon = 0.1\%$ error quantization. ISABELA accurately captures the physical phenomena as seen with the original data.

### Analysis of GTS Turbulence Datasets

GTS is a particle-based simulation for studying plasma micro-turbulence in the core of magnetically confined fusion plasmas of toroidal devices in nuclear reactors. We analyze
Figure 2.10: Plot of derived turbulence intensity values over 620 timesteps across different radial zones in XGC simulation.

the accuracy of ISABELA with respect to the original data, by comparing the Fourier spectrum of potential values in the poloidal dimension. We perform an FFT on each one of the ”circles” on a plane and get a radial variation of the spectrum on a 2D plane, and compute the relative error on the Fourier-transformed original and ISABELA-compressed values. While we have not included a visual analysis in this work, the spectrum plots for both the original data and ISABELA are equivalent, with little-to-no discernable difference between the two, suggesting the high accuracy of the results.

Analysis of FLASH Astrophysics Datasets

FLASH astrophysics code is a computational tool for studying thermonuclear reactions. To test the aggregate effect of pointwise error-bounded data reconstruction, we performed $k$-means clustering for various values of $k$ on the FLASH (and GTS) data sets. Table 2.7 shows the mis-classification rate of $k$-means on the compressed data compared to the raw, unaltered data, for various $k$. Here, mis-classification refers to the percentage of reconstructed data points that were assigned a different class than their corresponding raw data points. Randomized centroids were used, and the clustering ran for 100 iterations. The rate over all cases is extremely low on average, suggesting that the use of reconstructed data will not change the result of the $k$-means cluster analysis function by a meaningful degree.
Table 2.7: Mis-classification rate for 100-iteration $k$-means on the Flash and GTS datasets. Sample size is 4096 2-D points. $\epsilon$ represents the bounded error per data point.

<table>
<thead>
<tr>
<th>Num Clusters</th>
<th>$\epsilon = 0.1$</th>
<th>$\epsilon = 1.0$</th>
<th>$\epsilon = 0.1$</th>
<th>$\epsilon = 1.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.34%</td>
<td>0.28%</td>
<td>0.03%</td>
<td>0.25%</td>
</tr>
<tr>
<td>7</td>
<td>0.51%</td>
<td>0.42%</td>
<td>0.03%</td>
<td>0.31%</td>
</tr>
<tr>
<td>9</td>
<td>0.57%</td>
<td>0.46%</td>
<td>0.04%</td>
<td>0.36%</td>
</tr>
<tr>
<td>11</td>
<td>0.66%</td>
<td>0.56%</td>
<td>0.04%</td>
<td>0.40%</td>
</tr>
<tr>
<td>13</td>
<td>0.74%</td>
<td>0.60%</td>
<td>0.05%</td>
<td>0.45%</td>
</tr>
<tr>
<td>15</td>
<td>0.81%</td>
<td>0.67%</td>
<td>0.05%</td>
<td>0.49%</td>
</tr>
<tr>
<td>20</td>
<td>0.91%</td>
<td>0.77%</td>
<td>0.06%</td>
<td>0.57%</td>
</tr>
</tbody>
</table>

Analysis of S3D Combustion Datasets

The S3D code [34] performs first principles based direct numerical simulation (DNS) of reacting flows which aids the modeling and design of combustion devices. We analyze four variables from a time-dependent S3D simulation of a turbulent flame: temperature, $u$, $v$ and $w$ velocities across 50 timesteps. We compute the temporal averages and variances for these four variables at each of the 20 million grid points, and evaluate the relative errors with respect to the original data. In Table 2.8, we report only the global maxima of the relative errors for each of the quantities. The error in all of these quantities is $>1\%$ for only $0.38\%$ of the 20 million points. Since we compute relative errors with respect to the original data, we ignore those points in the field, for which both the original and the compressed data hold insignificant values (almost 0).

2.6 Related Work

Lossy compression methods based on spline fitting or Wavelets have been primarily used in the field of visualization, geometric modeling and signal processing. Very few studies applied such techniques when neither spatial nor temporal correlation of data can be directly exploited. Chou et al. [15] and Lee et al. [43] explored spline fitting for random data to optimize the location of control points to reduce approximation error. In contrast, we aim to transform the data to take advantage of the accuracy and easily-expressible qualities of splines.
Table 2.8: S3D Global maxima of absolute relative errors with the original data, for mean and standard deviation across 50 timesteps for 20 million points. For mean of velocities $u$, $v$, $w$, we consider values $> 3m/s$, and for standard deviation $\sigma > 1m/s$. For temperature we consider values with $\sigma > 5$ Kelvin (the peak variance in the domain is about $3.8E + 05K^2$).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>0.04%</td>
<td>3.7%</td>
</tr>
<tr>
<td>$u$</td>
<td>0.44%</td>
<td>1.6%</td>
</tr>
<tr>
<td>$v$</td>
<td>0.35%</td>
<td>0.7%</td>
</tr>
<tr>
<td>$w$</td>
<td>0.49%</td>
<td>1.4%</td>
</tr>
</tbody>
</table>

Extensively used lossy compression techniques like Discrete Cosine Transform (DCT), and Discrete Wavelet Transform (DWT) utilize spatial properties inherent in scientific data to achieve compression. DCT and DWT transform the data into frequency and spatio-frequency domains respectively, reducing the number of coefficients required to capture the data in the transformed space. Compression is achieved by thresholding away coefficients with small values.

The robustness of the sorting pre-conditioner is evident from the above results obtained using ISABELA, which outperforms transform-based Wavelet reduction. As an alternate to $B$-spline based reduction after sorting, using transform-based techniques to compress monotonic data can be effective as well. In a monotonic set of values, the spatial correlation of data is maximized, and both DCT and DWT can be expected to provide comparable levels of accuracy. However, even as accuracy levels remain comparable to ISABELA, point-wise decompression will no longer be possible. Query-driven analytics require local decompression in order to be efficient, which is lost when performing multi-level transformations on the data.

Lossless compression techniques [11, 32, 44, 51] have been recently applied to floating point data. Unlike most lossless compression algorithms, the techniques presented in [11, 44] are specifically designed for fast online compression of data. Lindstrom and Isenberg [44] introduced a method for compressing floating-point values of 2D and 3D grids that functions by predicting each floating point value in the grid and recording the difference between the predictive estimator and the actual data value. They also provide
the option of discarding least significant bits of the delta and making the compression lossy. However, the number of significant precision bits that can be saved is limited to 16, 32, 48, or 64 for double precision data. When applied to a one-dimensional data from GTS simulation, storing only 16 significant bits provided a compression of 82%, which is comparable with ISABELA’s, but more than 75% of points had per-point relative error of over 1%. By storing 32 bits, the per-point relative error was found to be within 0.1%, but the compression rate achieved (58.2%) was 13% less than ISABELA’s. Moreover, like other lossless algorithms, location specific decoding is not possible.

Other lossy compression techniques use variants of transform-based reduction, followed by some form of data quantization to compress the data sets that are fed as input to visualization tools. However, visualization community focuses on providing multi-resolution view-dependent level of detail. The error rate tolerated with lossy compression techniques on data used for visualization tend to be higher when compared to the data used for analysis. Hence, very little work exists that accurately compresses non-image or seemingly random data, even outside the scientific community. In fact, to the best of our knowledge, ISABELA is the first approach to use $B-$spline fitting in the context of reduction for data that is essentially random.

2.7 Conclusion

This chapter describes ISABELA, an effective in-situ method designed to compress spatio-temporal scientific data and perform analytical operations over the compressed data. The ISABELA compression starts by partitioning the data into small windows, and applying a sorting pre-conditioner, which significantly improves the efficacy of cubic $B$-spline spatial compression. It also exploits temporal patterns prevalent in scientific datasets, applying a $\Delta$-encoding of the higher-order differences in index values to further reduce index storage requirements. The indexing scheme built on top of the ISABELA data presents a storage-efficient format that accelerates query processing on heterogeneous architectures.

On 20 scientific datasets, ISABELA provides excellent approximation and reduction. Not only does it consistently outperforms Wavelet transform technique, but also delivers better performance, in terms of both compression ratio and accuracy. By capturing the relative per point errors and applying error quantization, ISABELA provides over 75%
compression on data from XGC, GTS and Flash simulation applications, while ensuring 99% accuracy on almost all values. Furthermore, several analytical operations such as correlation and query-driven processing benefit from quick approximate solutions that can be obtained by operating over ISABELA-compressed data. The storage-efficient solution over error-bounded compressed data, leads to accurate results on analytical operations over XGC and GTS simulation data sets (> 99% at $\epsilon = 0.1\%$) when compared with the original data. The ISABELA-compressed data and its parallel storage framework are thus ideally suited for scientific data analytics and visualization routines.
Chapter 3

ISABELA-QA: Query-driven Analytics with ISABELA-compressed Extreme-Scale Scientific Data

3.1 Introduction

Increasingly complex simulation models can now use high-end computing to simulate the dynamics of relevant biophysicochemical processes at the necessary level of detail. Along with this opportunity come new challenges. As the complexity of scientific data is growing, the demands for sophisticated, exploratory data analytics are increasing. Yet, the data size, already outpacing tera- and peta-byte scales, is becoming a rate limiting factor to our understanding of complex scientific phenomena hidden in this data. In response to this need, we present ISABELA-QA, a technology for performing query-driven analytics over ISABELA-compressed data (Chapter 2) rather than full-size raw data.

Fundamental differences in data context and access patterns exist among the simulation and exploratory data analytics steps that place some unique I/O requirements on the analysis and visualization of large-scale simulation data sets. The data generation process of space-time simulation proceeds in a local manner, from one time step to the next, and requires the context of only two time steps, while storing data for only one time...
step at a time on the disk. In contrast, data analysis and visualization steps often require the full context of the available data, not just a single time step. In fact, simulations that are driven by local space-time relationships are largely performed with the purpose of discovering or explaining global and large-scale space-time relationships through analysis and visualization [28].

Thus, the full context analysis and visualization software become tremendously data-intensive, creating a potential bottleneck in the entire scientific discovery cycle. Scaling up to full context faces severe constraints of computer memory, forcing the results of multiple data analysis and visualization steps to be stored on the disk. Moreover, such a data exploration process is interactive in nature, requiring almost real-time I/O access rates. Thus, the analysis and visualization software rely on high-performance I/O much more heavily than simulations in order to provide scientists with an interactive environment for the full context data exploration. However, it is a well-known phenomenon that I/O access rates have not kept pace with high-performance computing performance as a whole. For this reason, new paradigms are needed to support the unique needs of analysis and visualization for extreme-scale simulation applications, especially in the full context.

It is often the case that scientists have some prior knowledge about the regions of interest to look at. For example, the fusion scientists who aim to understand the nature of plasma turbulence could formulate their analysis question as one of the follows: (1) Calculate the time correlations of turbulence intensities between different radial zones \((0.1 < \psi < 0.15; 0.3 < \psi < 0.35; 0.5 < \psi < 0.55; 0.7 < \psi < 0.75; 0.9 < \psi < 0.95)\) or (2) Calculate the turbulence intensity flux at each radial zone and their time correlations. Likewise, climate scientists that aim to understand what factors may contribute to extreme events, such as hurricanes, might be looking for (1) teleconnections (or long-memory affects) that induce pressure gradient along the east-west direction over the ocean that may cause propagation of hurricanes from their embryonic Western Africa region toward North America or (2) correlations between sea surface temperature anomalies and hurricane activity in North Atlantic region.

Arguably, knowledge priors-guided data analytics is a paradigm that could not only reduce data to its practical size but also offer scientists a more appealing functionality for better understanding of their hypothesis-driven experiments. This brings yet another question: what is the proper way for scientists to express their knowledge priors about their spatio-temporal data \(D\) defined by multiple variables \(V\) that evolve in some space
region \( R \) over some time period \( T \)? One way is to leverage database query semantics. Namely, let the analytics operator \( A \) be applied to the data subset \( D_q = (V_q, R_q, T_q) \subseteq D \) generated in response to a database query defined as follows:

\[
\text{SELECT} \quad \{ R_q \subseteq R, V_q \subseteq V, T_q \subseteq T \} \\
\text{FROM} \quad \{ I_V \} \\
\text{WHERE} \quad \{ R_c \subseteq R, V_c \subseteq V, T_c \subseteq T \}
\]

(3.1) (3.2) (3.3)

To ensure fast query processing, a typical approach, employed by database community, takes advantage of the index \( I_V \), such as \( B^+ \) trees, generated for variables of interest (see Equation 3.2). While being a “blessing” for efficient query processing, indexing is, arguably a “curse:” the index size is often 100%-300% of the original data size [72], which is a significant bottleneck for storage-bound, extreme-scale applications.

To alleviate this problem, bitmap indexing, called FastBit, has been maturing into a valuable technology due to its attractive features: (a) Its compressible bitmap index reduces the index size to 30%-170%; (b) It allows for query processing over the compressed index without the need for index decompression; and (c) Its bitwise logical operations enable fast Boolean query operations (see Section 3.2 for more details on FastBit). Due to these properties, FastBit has been successfully applied to a number of scientific applications [50, 74], and recently has been explored for accelerating scientific visualization such as isosurface rendering and for basic statistics with histograms [35].

From database perspective, FastBit is the state-of-the-art indexing technology for scientific data sets. However, from data analytics perspective, FastBit, for a number of reasons, is becoming a rate limiting factor. By design, FastBit has been optimized only for processing region-centric queries, namely for \( R_q \neq \emptyset \) but \( V_q = \emptyset \) (see Equation 3.1), with variables constrained by their range values (i.e., \( 50 < \text{Temperature} < 70 \) & \( 250 < \text{Pressure} < 850 \)), namely for \( V_c \neq \emptyset \) but \( R_c = \emptyset \) (see Equation 3.3). Yet, for data analytics, the knowledge of which points belong to the region \( R_q \) of interest is not sufficient. In fact, it is the actual values of the variables associated with those points are of primary importance. Analytics operators \( A \) could be highly sensitive if the quantized, or binned, values are used instead of their actual values (see Section 3.4.1). Thus, in order to be effective, query-driven data analytics must be optimized for variable-centric
queries, namely for $V_q \neq \emptyset$.

In order for indexing technology, such as FastBit, to support variable-centric queries ($V_q \neq \emptyset$), they have to keep the copy of the original data. Given the index size to be comparable with the size of the data, we are roughly talking about doubling the amount of data, which is an expensive proposition to the extreme-scale applications. In addition, their currently supported APIs and implementations for retrieval of the actual values for the identified points in the region of interest could either induce random I/O disk access or a full disk scan, which obviously defeats the purpose of having such an index in the first place.

Therefore, it is probably safe to conclude that, starting “from scratch” and designing the technology specifically optimized for query-driven data analytics is the emerging need for storage-bound, extreme-scale applications. To this end, we propose ISABELA-QA that offers the following advantages:

- It operates on the data compressed by our ISABELA technology to approximately 15%-25% of its original size.

- Its index is built on top of the metadata generated by ISABELA unlike traditional indexing approaches that are built on top of the original data. As a result, its index size is no more than 3% vs. 30%-300% of the original data.

- Compared to FastBit, ISABELA-QA’s performance for region-centric queries is comparable or faster, even in the serial processing regime.

- Compared to sequential disk scan and to FastBit, ISABELA-QA’s performance for variable-centric queries is orders of magnitude faster in a serial processing mode.

- ISABELA-QA also supports parallel, multi-node, multi-core, and multi-GPU implementation that is scalable due to its highly localized and embarrassingly parallel data processing regime.

In summary, ISABELA-QA, unlike existing state-of-the-art technologies, only requires $< 30\%$ of the original data storage without the need for keeping the copy of the original data, thus offering not only overall eight-fold or more data storage reduction vs. existing query processing methods but also multi-fold speed-ups for both region-centric and variable-centric queries.
3.2 Background and Related Work

Search and query processing operations on traditional database systems like Oracle, MySQL, DB2, and FastBit involve the use of indexing techniques that are usually variants of either Bitmap indexes or B-trees [23]. While these techniques are effective in speeding up query response times, they come at the cost of a significantly heavy weight index management scheme. Indexing with B-trees, which tend to be more suitable for transactional databases that require frequent updates, is observed to consume storage that is $3 - 4$ times the size of the raw data [72]. Scientific data, which is typically read (or append) only, are better served with bitmap based indexing techniques [36] that provide faster response times with lower index storage space of about 2 times the base data.

While there are numerous technologies that use variants of bitmap index, we primarily focus on FastBit, a state-of-the-art bitmap indexing scheme, that is used by a number of scientific applications [50, 74] for answering queries. Bitmap indexing techniques traditionally employ any combination of 3 tasks: binning, encoding, and compression [37]. Binning maps variables to bins so that variables of a similar property or value are colocated for quick lookup. For example, given a bin $B$ that represents a range $[0, 5)$, for each record, a bit is used to represent whether the record falls into a bin or not. A bitmap vector is encoded using this technique for each defined bin for the table. Since, the space taken up by the bitmap vectors becomes unmanageable for large datasets, FastBit employs a Word-Aligned-Hybrid compression scheme [36], based on run-length encoding. This compressed representation not only decreases the storage requirement, but also allows FastBit to directly perform logical operations quickly on the compressed space, and compute partial results by just scanning the index. For those records which cannot be evaluated with the index alone, FastBit resorts to performing a read of the raw data.

3.3 Method

3.3.1 Overview

An overview of the end-to-end query processing is given in Figure 3.1. Given a query of the form in Equations 3.1, 3.2, and 3.3, a number of operations are performed (note that in an optimized implementation the order may not follow Figure 3.1, but the logical flow
is the same):

- Based on the variable constraint, determine which bins need to be accessed and whether the bounds on the bin match the query. See Section 3.3.3.

- Obtain spatial region information from the binned index file. This allows us to search areas of the data most similar to our query. If variables are not required as output, we can directly output spatial regions within bins that are aligned with the query bounds. See Section 3.3.3.

- For bins not aligned on query bounds and if variables are requested as output, load and uncompress quantized error values, and apply to variable values regenerated using the values’ respective B-spline coefficients, ensuring bounded, per-point error. See Section 3.3.3.

- Finally, for bins aligned within the query boundary, return the spatial-regions and/or variables directly. For bins not aligned within the query bounds, filter the variable values against the query bounds. See Section 3.3.3

Overall, our system trades extra computation on reconstructing values for a smaller storage footprint and much lower I/O costs. This plays well in HPC systems, which have an abundance of computational power and by which I/O is a bottleneck for large-scale data-intensive operations, especially in multi-user environments.

### 3.3.2 Data File Layout

There are three types of files used in our query processing engine; a spatial region file for each bin, a quantized error file for each bin, and a file listing the B-spline coefficients. A query requesting spatial region and variable values would determine the bins to read based on the variable constraints, then read the timestep-delimited data from the bins, finally loading coefficients based on compression windows hit by the query. The format of our bin-based index files are shown in Figures 3.2 and 3.3, and were developed with a number of goals in mind:

- Minimize the overhead storage costs,

- Minimize need for metadata storage in memory,
Figure 3.1: Overview of the query processing engine. Values in brackets represent optional pieces of the flow, such as variables being reconstructed for bins in $B_A$ when $V_q \neq \emptyset$. See Section 3.3.

- Minimize the number of seeks/reads for any particular query, and
- Organize the data to allow decomposition of a query into parallel workloads.

When considering overhead storage costs, there are two pieces of metadata: file offsets for timesteps and compression window information, such as the start position within the window and the number of elements in the window in a bin, giving context to the stored spatial regions/errors. The file offsets take insignificant space while the window information takes somewhat more space, requiring at most a datum for every window for every bin, though this is highly unlikely to happen. Still, the amount of data this metadata comprises is a small portion of the overall storage costs. For instance, we fix the window size to be 1024 elements, meaning, for each bin, the absolute worst case would include about $0.3\% - 0.5\%$ of the raw data size per bin, or about $6\% - 10\%$ for 10 region and error bins, though this case would only occur if every compression window is comprised of variables across the full range of values in the original data, which is unlikely based on our empirical observations.

The only mandatory memory usage for the file structure is the file offsets per bin, meaning that the number of offsets is of the order of the number of bins times the number of timesteps. Since we use coarse-grain binning, rather than the fine-grain binning of alternatives such as FastBit, the memory cost of storing the offsets is a small multiple of the number of time steps. In our experiments, we used 10 bins, for a total of 20 times...
the number of timesteps.

On a per-bin basis, assuming the timesteps requested are contiguous, I/O operations for query processing require a single seek and multiple read operations per bin, where each read loads an entire timestep’s worth of data stored in the bin. When the timesteps requested are non-contiguous, multiple seeks are required. In either case, the I/O costs are much lower than random or full sequential reads from the raw data.

In discussing scalability of the binning representation, it is important to note that file operations are non-conflicting since the query process is a read-only process, operating only on a single timestep within a single bin at a time. Therefore, the process can be made embarrassingly parallel by distributing the timesteps to read among different compute elements. However, it is important to note that the amount of data stored per timestep per bin is highly variable, preventing a regular striping, or partitioning, of the data across discrete storage elements. Each timestep could be padded to a regular length to enforce such regularity, though at a potentially large cost of additional storage space. For this reason, we do not explore padding here.

\[
\begin{align*}
<N \text{ number of timesteps}> \\
<File \text{ offset for timestep } i> \ (0 \leq i < N) \\
(Repeat \ 0 \leq i \leq N) \\
<W \text{ number of windows in timestep } i> \\
<Window ID \ i> \ (0 \leq i < W) \\
<Window offset \ i> \ (0 \leq i < W) \\
<Window length \ i> \ (0 \leq i < W) \\
<Window indices> \ (\sum_{j=0}^{W-1} \text{window length } j) \\
(End \ Repeat)
\end{align*}
\]

Figure 3.2: Bin-based index file format for spatial regions.

### 3.3.3 Query Decomposition

Equations 3.1, 3.2, and 3.3 show a prototype query. For simplicity, we assume a single variable index and single variable constraint. As the purpose of our system is to accelerate queries based on variable bounds, we also assume $R_c$ is empty. We will also focus on univariate queries, as multivariate queries will be explored and optimized in future works.
Figure 3.3: Bin-based index file formats for quantized errors. Each window tuple contains
the window ID, the window offset, the number of values within the window in the bin,
and the offset in memory of the uncompressed $k$-vector. *Each section is compressed
using the standard compression library zlib.

Bin Determination

Given the bound $x \leq v \leq y$, where $v$ is the variable in question, the bins can be split
into two sets, $B_A$ and $B_M$, or "aligned" and "misaligned" bins, respectively. Aligned
bins are those by which the bounds of the bin are contained within the bounds of the
value constraint. Thus, if the variable values are not required as output, only the spatial
regions need to be read from these bins, since the regions are guaranteed to correspond
to values within the query. The misaligned bins have bounds not contained within the
query bounds, so there may be spatial regions which fail to satisfy the query constraints.
Since determining which of these points violate the constraints in advance, the values
corresponding to the spatial regions must be reconstructed, regardless of whether they
are needed in the output.

For example, consider bins $B_1, B_2$ such that variables within $B_1$ fall within the range
$[-1.0, 0.0)$ and variables within $B_2$ fall within the range $[0.0, 1.0]$. A request for variables
in the range $[-2.0, 2.0]$ would lead to $B_A = \{B_1, B_2\}$ and $B_M = \emptyset$. A request for variables
in the range $[-2.0, 0.5]$ would lead to $B_A = \{B_1\}$ and $B_M = \{B_2\}$, as some variables in
$B_2$ may not fit in the range of the query.

Spatial Region Retrieval

Once the bins have been determined, the index offsets in memory are used to perform a
seek to the timestep(s) in file, as described in Section 3.3.2. Simply enough, the window
IDs, the starting offsets within each window, the number of elements of each window,
and the window indices are fetched and unpacked. With a fixed window size $W$, offsets, the number of elements, and the window indices can be packed into $\lceil \log(W) \rceil$ bits each. If the variables aren’t required as output, then all indices in the bin set $B_A$ are returned immediately, following a mapping from window index back to spatial region.

Variable Retrieval/Reconstruction

For the bin set $B_M$, and additionally $B_A$, if the variables are required as output, the variables corresponding to the spatial regions within the bins must be reconstructed. First, the B-spline coefficients are read into memory and the variables for the relevant windows are regenerated. Second, the compressed error data is fetched into memory and decompressed. The window information is unpacked and the error-level-vector (i.e., the number of quantized error values per point) and quantized errors are decompressed. The quantized errors are applied to the regenerated variables, yielding the error-bounded variables. From here, the spatial regions and reconstructed variables can be returned for bins in $B_A$, but for bins in $B_M$, a filter operation is performed on the reconstructed values, throwing out those that are not within bounds. Note that this filtering can be performed during the reconstruction process, utilizing the sorted order of the compression window to stop regeneration once the values in each window go out of bounds.

3.3.4 Accelerating Query Processing

While the use of variable-based binning minimizes the number of elements examined during the execution of a variable-bounded query, there is a large degree of overhead for generating values and performing disk I/O, especially for large datasets and query sizes. First, since we are reading packed or compressed data, the I/O portion may not be a bottleneck as the amount of data we are regenerating is much larger than the data we are reading from disk. Second, there may still be a large number of elements to read from memory under various constraints, increasing I/O cost. This cost can be exacerbated by requesting data in non-contiguous timesteps, thus increasing the seek costs, a significant concern when working on distributed filesystems. The following sections describe strategies for mitigating these costs.
Multi/Many-core Optimization

Regeneration of variables from a window is a data-parallel method, where each point in the window can be computed independently of the others. This point-by-point independence is enabled by the use of B-splines in modeling the sorted-per-window data; given the B-spline coefficients, any arbitrary point within the spline can be generated. This leaves two simple methods of parallelizing this operation. The points can be considered a data stream where operations on each datum is performed based on coefficient values. This data-parallel approach caters to GPU computing, but incurs CPU-GPU transfer overhead. Another possibility is splitting the windows among a number of CPU threads, taking advantage of multiple cores in the CPU, while keeping the data localized in memory.

Error decompression and application can also be parallelized, though with a few considerations. Because of the decompression method used, parallelization of the decompression itself is not possible unless we break each compressed section into a different number of streams, which can lower the compression rate. The application of errors to the variables can be simply parallelized, though. However, a GPU-based implementation of error application has high overhead costs, since the amount of computation per value is much lower compared to regeneration.

Two implementation strategies are used to achieve these parallelizations: GPU computing based on Nvidia’s CUDA programming model [1] and OpenMP [20]. In OpenMP, the parallelized reconstruction is as simple as putting a precompiler directive in front of the B-spline regeneration loop, dividing the set of windows to reconstruct among threads. Better yet, threads can be divided on temporal contraints, parallelizing all non-I/O aspects of the query processing. On the GPU, a similarly simple approach is taken to parallelize by window, though with additional transfer steps. First, the window IDs, range values, coefficients, and per-window offsets in the output stream, obtained by a prefix sum on the number of variables in each window to be regenerated are transferred into GPU main memory. Each thread block takes a different window, and threads compute each value in its specified range in strides. Once the kernel finishes, the regenerated variables are transferred back to the CPU. Note that the transfer to the GPU is a low-cost operation compared to the transfer of values back, which helps reduce the round-trip overhead associated with a discrete graphics card.

While parallelization of the reconstruction step is simple, overhead costs and load-
balancing issues necessitate determining under what conditions the acceleration is actually useful. For instance, the amount of work performed by smaller queries may be smaller than the overhead required to multi-task and handle output. When considering performing reconstruction on the GPU, transfer overhead is an important concern. Furthermore, a poor distribution of points to each computing unit may cause some computing units to take the majority of work while others are idle. These scenarios are discussed in Section 3.4.5.

Multi-node Optimization

Multi-and-many-core acceleration of the compute-bound portions of our query processing can efficiently accelerate the reconstruction process, but for larger and larger problem sizes, an implementation is needed that utilizes much more computing power, especially for queries that require regeneration of billions of points. Taking a distributed approach is just as simple as in the OpenMP/GPU approaches, though in this case the parallelization is performed across temporal constraints rather than per compression window. Parallelizing by bins is another approach, but the coarse-grain binning performed by our system does not allow for scalability in this regard. Parallel I/O can be utilized and is discussed in Section 3.3.2; each process computes the spatial regions and values on a single timestep. Timesteps corresponding to a query are distributed by the root process.

There are a few important issues that need to be addressed that are not present in the other implementations. First, the distribution of points between timesteps in a bin can vary to a large degree; some may have no variables at all, where others may be fully populated. A distribution of timesteps must then be modified to promote a load-balanced distribution of work. Another possible problem is the handling of generated points, which may need to be aggregated onto the root process. For very large queries, it is likely that the transfer of such a large number of points will become a bottleneck, negating any benefit gained from a large core count.

To help solve these problems, the timesteps are sorted by their size in file before being distributed to the processes in strides. While this increases seek costs per process, the penalties for having a poorly load-balanced compute distribution are much steeper, and it will be shown that the additional I/O costs are not significant. In the aggregation process, a gather operation is performed at the end of the query process for each bin, transferring spatial regions and variables by timestep for the head process to output.
We split the gathers up into groups of timesteps under the assumption that the entire output space may not be able to be held in memory. In future works, we will examine methodologies for efficient shared-file writes.

3.4 Results

3.4.1 User-Centric Perspective

Quantization Effect on Point-by-Point Values

In the previous chapter, we analyzed performance of ISABELA against wavelet-transform-based data reduction techniques and showed that ISABELA consistently exhibited superior error rates on a wide range of datasets [54]. We compare the accuracy of ISABELA with FastBit’s histogram-binning, which maps the dataset into pre-defined binning intervals, but produces only spatial regions in response to queries. Thus, disk I/O on raw data must be used to filter out regions in misaligned bins, or otherwise, retrieve the variable values. To avoid the random disk I/O and allow for reasonable comparison, we emulate quantization by returning the midpoint of the bin containing a spatial region. In Figure 3.4, we analyze the effect that the size of the binned variables has on the accuracy of quantization. Clearly, a fine-grained binning is required to return quantized values that are reasonably accurate when compared to the original, but this
comes with a heavy trade-off, with the index size becoming as voluminous as the raw data. Moreover, the accuracy of the quantization becomes directly dependent on the distribution of values as well.

Quality of Query-driven Analysis

In this section, we analyze the quality of query results obtained with ISABELA-QA. Figure 3.5 shows the variation of precision in the retrieved result over varying query selectivity sizes. Query selectivity here refers to the ratio of the amount of data returned by the query to the actual data, and precision refers to the number of “relevant” data points retrieved divided by the total number retrieved. Naturally, when the user-defined relative error parameter ($\epsilon$) is decreased when storing the data, the quality of query results on ISABELA-compressed data becomes more accurate. On a query of the form $variable < value$, increasing the selectivity results in precision value $\approx 1$ (Figure 3.5). This is because when the data follows a normal distribution, the absolute value of errors includes a larger number of points at the tails, which results in lower precision or larger false positive rates. To ensure false negatives do not exist, user-defined queries are rewritten on ISABELA-compressed data to include the range $\pm \epsilon$.  

Figure 3.5: Precision of query results obtained on ISABELA-compressed 3.2GB GTS potential data.
3.4.2 Experimental Evaluation

We performed our experiments on the Lens cluster on ORNL that is dedicated for high-end visualization and data analysis. Each node in the lens 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. In the next sections, the term *Query Selectivity* is taken to mean the percentage of the full database returned by a query.

3.4.3 End-to-End Performance: Variable-Centric Queries

Figure 3.6 show the performance of ISABELA-QA when compared to FastBit and a sequential scan of the raw data when retrieving variables along with region points for varying query selectivity values. *Query selectivity* refers to the amount of data returned by a query, as a percentage of the entire database. While it takes several hours to serially scan through all the partitions of the data with the other techniques, ISABELA-QA returns the answer in less than 10 minutes on a terabyte of data in serial. The I/O operations contribute to less than 10% of the actual query processing time, and due to the clustering of the data, a small number of file I/O or seek operations need to be performed. The additional monotonicity property of data within each window in ISABELA-QA ensures that query ranges can be found efficiently. As a consequence, not all points need to be interpolated, which again reduces the response times. While the CPU time used...
by FastBit is less than 1%, I/O times dominate for almost all queries where fetching voluminous data over a distributed filesystem, and scanning through the data to retrieve the values for result set drastically slows the response time. The sheer volume of data processed renders caching strategies widely employed by database management systems ineffective. In fact, as the query selectivity increases, processing over any indexing scheme induces an additional overhead, and it ends up being slower than a sequential scan.

ISABELA-QA can reconstruct data in parallel on multiple levels of spatial and/or temporal resolution. While currently, no mechanism is in place to perform load balancing with OpenMP, even a simple multi-core parallelization with OpenMP on the temporal resolution, provides more than a 10-fold decrease in computational time, and an 8-fold decrease in the overall response times compared to serial processing, thus reducing the original 10-minute time with serial ISABELA-QA processing to less than a minute with a single node, multi-core query processing of a terabyte of data.

3.4.4 End-to-End Performance: Region-Centric Queries

For region-centric queries, FastBit needs to hit the disk only to evaluate those points that fall into the misaligned bins. The maximal number of points that would need to be evaluated for FastBit with a single misaligned bin is $N/N_{\text{bins}}$, where $N$ is the total number of points, and $N_{\text{bins}}$ is the total number of bins. It becomes clear that the number of bins used, $N_{\text{bins}}$, plays a definitive role in determining the time taken to respond to region queries. In Figure 3.7 we see that increasing the number of bins, reduces the number of
file seeks that need to be performed in the worst case. But using a larger number of bins quickly deteriorates the performance of I/O bound value-centric queries. In the case of ISABELA-QA, the monotonicity property is again exploited in its entirety to process region-centric queries. Region-centric queries only involve the start and end regions to be identified in each window. A binary search on this ensures that no more than 20 points need to be reconstructed, within a single window in a bin. Once these end points are computed, only the corresponding regions need to be retrieved. Although this does not match FastBit when the number of points returned is extremely low, the performance is consistent even when larger regions need to be retrieved.

### 3.4.5 Performance Analysis

For our query processing engine, to determine the performance characteristics, we first compare the serial performance of each component, and also compare the I/O and compute costs. Once baseline characteristics have been established and bottlenecks identified, we will examine how the various acceleration strategies discussed in Section 3.3.4 affect the overall run time and the contribution of each component. [Remind that query selectivity refers to the amount of data returned by a query, as a percentage of the entire database.]

![Figure 3.8: Performance of computational components of ISABELA-QA query processing over varying selectivity.](image)
Serial Component-Based Analysis

Summarizing the various components of processing a single query, the I/O costs of query processing includes loading B-spline coefficients, spatial regions, and compressed quantized errors in their respective bins. The compute portion includes B-spline regeneration of values and the decompression and application of quantized errors to those values.

For the compute portions of the query process, Figure 3.8 shows the contribution of each component as a percentage of the running time (Spatial region identification refers to unpacking the compressed window information and the spatial regions). There are several items of interest here. First, for very sparse queries, decompression of error values is the dominant cost, as for each timestep requested, the entire set of error values is decompressed while only a few of the values are actually used, due to cutting off out-of-bounds regeneration. As queries get larger in selectivity, however, reconstruction takes the lion’s share of the processing time due to the much larger number of points. The error application time increases as well to approximately match the decompression time. Overall, this seems to suggest that breaking up the compression of errors into multiple streams would do much to speed up sparse queries, while parallelizing the reconstruction would be useful everywhere, but especially for larger selectivity values. Also, it should be noted that when parallelizing by timestep, all of these measurements are parallelized.

Figure 3.9 shows the overall compute time compared to the overall I/O time as a percentage of running time. Note that we observed little difference between the various
operations on different files, as they are accessed in a similar fashion. Here, we see that regardless of query size, the I/O costs associated with query processing is very low, which makes sense considering the compressed form of the data being read and the need to decompress and reconstruct values point-by-point. Obviously, reducing the computation time is of paramount importance in speeding up the query processing.

Analysis of Acceleration Strategies

![Figure 3.10: A comparison between serial, CPU-parallelized, and GPU-parallelized B-spline regeneration. 8 OpenMP threads were used in this particular plot, corresponding to two quad core Intel Xeon X5355 processors.](image)

It has been established that the compute portion of ISABELA-based query processing is by far the most significant cost. Fortunately, this means that conventional compute-based acceleration strategies can be utilized effectively, especially given the attractive parallelization properties of our query processing method discussed in Section 3.3.4. We look at the effectiveness of acceleration strategies in single-and-multi-node environments.

Accelerating B-spline regeneration in a single-node environment is shown in Figure 3.10. Note that the OpenMP implementation shown was parallelized per-timestep, rather than the within-timestep parallelization of the GPU method, and that regeneration times for both parallelizations for OpenMP are identical. At the fine granularity of
Figure 3.11: Speedup of queries requesting spatial regions only and optionally variables. 16 OpenMP threads are used, on four quad-core processors on a single node.

generated points shown, there are large performance increases for both the OpenMP and the GPU implementations. OpenMP performs very closely to the GPU method, making both appealing candidates to mitigate bottlenecks resulting from the need to regenerate large numbers of values. Furthermore, the serial CPU method is never a clear winner except for an extremely small number of points, so the need to decide which implementation would be right for any query is not necessary.

Figure 3.11 shows the performance of overall query processing time with a serial process vs. a timestep-parallelized OpenMP implementation. The uneven distribution of spatial regions and variables over the timesteps prevents optimal scaling, but regardless, good scaling is seen, suggesting that we can minimize the compute portion of query processing by throwing more threads/cores at it, at least for queries asking for variable values. For the region-based queries, we see a much smaller speedup, as the computational cost is smaller overall (that is, only the start and end of a window which fit the bounds need to be found by, for example, binary search, reducing the reconstruction cost significantly).

For distributed query processing, Figure 3.12 shows the portion of work performed under the compute, I/O, and aggregation parts of the algorithm using only 32 GB of data and 1% query selectivity. As it can be seen, the compute portion is highly efficient with respect to the number of cores, the I/O portion remains roughly the same as the irregular file structure is unable to utilize the underlying file system efficiently, and the gather portion increases rapidly, quickly overtaking the two. While not explored further
here, in future works, we expect to explore multiple methods of removing this limitation. The simplest fix is having each process output data to their own file, performing aggregation as a post processing step. However, this merely moves the problem elsewhere. A more intelligent solution would involve either broadcasting the amount of output to each process, then writing to a shared file in each process’s designated location, or overlapping transfer of output, computation, and I/O to hide the transfer cost. The latter solution, however, would only work when there is enough work per process to effectively hide the transfer. If the aggregation can be performed efficiently, which is promising given the proposed solutions, then distributed query processing can be efficiently performed on terascale and beyond datasets.

3.5 Conclusion

As scientific datasets continue to grow at a rapid rate in both size and complexity, the need for data analytics over reduced data is increasing, especially for full context data analysis and visualization. Yet, naïve approaches over randomly sampled data are either I/O-bound due to the significant random disk access overhead or suffer from inability to discover unusual, extreme or rare events (i.e., needle in a haystack effect). Arguably, a paradigm shift, such as the one advocated in this chapter, is a promising strategy to bridge the growing gap by enabling query-driven data analytics over the ISABELA-compressed data. The resulting technology, called ISABELA-QA, is the first and the
only approach that offers memory-and-storage light-weight solution with embarrassingly parallelizable characteristics for efficient processing of not only the spatial region-centric queries but also the variable-centric queries that are necessary for accurate and robust data analytics of scientific spatio-temporal multi-variate data. Unlike state-of-the-art scientific database management solutions, such as FastBit, ISABELA-QA offers both the accurate, when compared to quantization, and multi-fold faster, when compared to even serial implementation, retrieval of the variable values in response to range queries. In addition, its initial prototype of parallel, multi-node, multi-core, and multi-GPU implementation is a promising solution for the I/O-and-storage-bound applications at extreme scale.
Chapter 4

In-Situ Indexing and Aggregation

4.1 Introduction

As described earlier, the concept of in-situ processing, or processing data at application run time and in application memory, is one of increasing importance. The traditional approach of performing data processing, analysis, etc. as a post-processing step is becoming a rate-limiting factor as application data sizes increase faster than I/O capabilities. Recent research has been investigating the design space and implications of in-situ processing and data staging frameworks to facilitate this model [2, 3, 4, 80, 66].

While the concept of in-situ processing has been realized in such areas as visualization [47, 78, 65], analysis frameworks [80] and compression [56, 54] (see Chapter 2), in this chapter, we focus specifically on index generation. Such indexing enables the acceleration of tasks, such as exploratory and query-driven analysis, that may not themselves be amenable to in-situ processing, thus indirectly reducing time-to-analysis. This approach to supporting query-driven analytics for large-scale data has only just begun to be studied. Recently, the bitmap indexing technique FastBit [52, 74] has been applied in parallel with FastQuery [17, 16, 12] and extended to demonstrate indexing in an in-situ context [38].

However, in order to extend in-situ indexing to the production context of high core count application runs, several challenges must first be overcome. Most index generation processes are both computationally expensive and storage intensive, incurring significant processing and I/O overhead. These are opposed to one of the central goals of in-situ computation: to minimally disturb application run time. Furthermore, as indexing in a
global context is prohibitively expensive due to the need for global coordination, current methods of index generation produce fragmented indexes across compute resources, which significantly increases query response time. Related to these overheads is the memory-intensive nature of indexing, placing hard constraints on the memory overhead of indexing and limiting the degree of aggregation that can take place.

To address these challenges, we propose a methodology for Data Indexing and Reorganizing for Analytics-induced Query processing (DIRAQ). The following contributions enable us to make inroads towards a storage-lightweight, resource-aware data encoding technique that incorporates a query-efficient index:

**Storage-lightweight, Query-optimized Data Encoding** We describe an encoding technique that converts raw floating-point data into a compressed representation, which incorporates a compressed inverted index to enable optimized query access, while also exhibiting a total storage footprint less than that of the original data. We exploit the spatio-temporal properties of the data by leveraging our previous work with ALACRITY [33], augmenting it with a highly-compressed inverted index using a modified version of the PForDelta compression [81] algorithm.

**Scalable, Parallel Data Reorganization** For fixed-size groups of processes, we “de-fragment” indexes to optimize query performance by developing an in-network aggregation and merge technique tailored to our encoding, which distributes the computation equally among all compute cores in the group and allows arbitrary selection of aggregator cores to gather/write the resulting data. This way, we avoid the pitfalls of serializing the encoding process at various stages.

**Resource-aware Aggregation** We additionally make our group-wise indexing resource-aware, dynamically learning optimal data paths and choices of aggregators through per-group neural network modeling that supports online feedback. The optimization space for the model is constrained by the available memory, ensuring memory constraints are not violated.

Our proposed method shows promising results on 9 datasets from the FLASH astrophysics simulation [25] and 4 datasets from S3D combustion simulation [14]. Our encoding reduces the overall storage footprint versus the raw data by a factor of 1.1–1.8x, and versus FastQuery-indexed data by 3–6x. Our scalable reorganization and aggregation method combined with our encoding allows up to 6x to-disk throughput improvement.
compared to MPI-IO on the raw data. Finally, query performance on our defragmented indexes is improved by up to 10x versus FastQuery-generated bitmap indexes.

4.2 Related Work

In this section we cover previous work related to in-situ processing, in-situ indexing and aggregation strategies for I/O.

The onset of petascale and exascale computation has seen a significant growth in works that encompass simulation-time processing relating to in-situ and in-network processing [8, 47, 78], along with several data staging systems such as JITStaging [2], DataStager [4], DataTap [3], PreDatA [80] and GLEAN [66] that explore movement of simulation data to co-located clusters for processing. The DIRAQ pipeline has been carefully designed to complement staging-driven approaches for generating indexes; parts of the index merging process can be offset to staging routines, but is not the focus of this study.

Distributed and parallel indexing itself has been well researched in the database community. The majority of the indexing techniques are variants of the commonly used B-Tree indexing technique, which have been shown to have sub-optimal performance for many workloads over read-only scientific datasets, when compared to other techniques such as bitmap indexing [71]. The parallel indexing scheme FastQuery [17, 16, 12] and subsequent in-situ work [38], which extend the WAH-compressed bitmap indexing method FastBit [52, 74], share the same overarching goal as DIRAQ. However, DIRAQ differs in that it utilizes in-situ index aggregation over a larger spatial context, instead of concatenating indexes from each core, making it more suitable for analytics, and because it explicitly addresses issues such as including network aggregation and limited I/O throughput.

In contrast to a post-processing context, performing indexing in-situ demands special attention to scalable I/O, an area with much prior work. MPI collective I/O is the canonical approach to this problem, which typically incorporates a two-phase I/O technique [64] to aggregate data into fewer, larger requests to the filesystem. This principle has been refined in various ways, including a 3-phase collective I/O technique with hierarchical Z-order encoding [39], and pipelined aggregation enabling the overlap of computation and threaded I/O [27, 26]. However, while these approaches are well-suited to a variety
of common I/O patterns, indexing introduces an irregular access pattern. To overcome this, we opt for a customized I/O aggregation strategy that includes in-network merging of core-local encoded data.

As for optimizing the I/O aggregation process, recent work has been performed on auto-tuning the number of aggregators involved in MPI collective I/O [13]. Depending on the amount of data written out with each group, they either merge or split process groups (indirectly changing the aggregator ratio) to better utilize the available I/O bandwidth. In this study, we group compute processes that belong to the same processor set (pset) since I/O forwarding is done on a per-pset level. While they use process mapping based on topology, we employ the aggregator placement to be topology-aware as well. Additionally, our method tunes aggregators within a group, rather than changing the underlying group size.

4.3 Background

4.3.1 ALACRITY - Indexing for Scientific Data

We use our previous work with ALACRITY [33] as the starting point for indexing in DIRAQ. ALACRITY is a storage-lightweight indexing and data reduction method for floating-point scientific data that enables efficient range query with position and value retrieval. Specifically, ALACRITY is optimized to identify positions satisfying range conditions (e.g. “temperature > 2500”) and efficiently retrieves the values associated with those points. It achieves this by utilizing a byte-level binning technique to simultaneously compress and index scientific data. Because ALACRITY integrates data reduction with indexing, it exhibits a much lower storage footprint relative to existing indexing methods. For instance, while ALACRITY’s total footprint (data + index) is consistently about 125% of the raw data for double-precision datasets, a typical FastBit [70] encoding over the same data requires ≈ 200%, and a B+-Tree may require more than 300% [71].

The key observation in ALACRITY is that, while floating-point datasets have a large number of unique values, the values are still clustered and do not span the entire floating point domain. If we examine the most significant $k$ bytes of these floating point values (typically $k = 2$), this value clustering translates into a list with much lower cardinality. This is because the IEEE floating point format defines the highest $k$ bytes (which we denote as high-order bytes) to contain the sign bit, exponent bits, and most
significant mantissa bits of the value. The high-order bytes will therefore exhibit lower
cardinality than the low-order bytes, which typically contain much more variation and
noise.

ALACRITY leverages this observation by binning on the high-order bytes of the
data. Because the exact high-order bytes are stored as bin “header values,” this inform-
ation does not need to be repeated for each value, and so the data is substantially reduced
by storing only the low-order bytes for each datapoint. As a property of the floating-point
format, each bin contains points belonging to a single, contiguous value range. Therefore,
by properly ordering the bins, range queries can be answered by reading a contiguous
range of bins in a single, contiguous read operation, significantly reducing the number of
seeks necessary to support value retrieval.

However, because the binning operation rearranges the values, an index is required
to maintain the original ordering. In the original paper, two alternatives were explored:
a “compression index” and an inverted index. The compression index was shown to be
effective in providing data compression, but is not appropriate for querying, and so we do
not consider it here. The inverted index, while larger, is still lightweight, with a storage
requirement of only 50% of the original data size for double-precision data, and is effective
for querying.

While ALACRITY works well in the context of serial indexing, we must develop
additional methods in order to support parallel indexing in DIRAQ. In particular, al-
though ALACRITY’s total storage size is notably smaller than previous methods, it
still represents a non-trivial I/O overhead beyond raw data, and would be expensive if
applied as-is for in-situ indexing. Additionally, index merging was not previously con-
sidered, as ALACRITY operated using full-context data. However, in this study, we
analyze the end-to-end performance of indexing, starting from core-local data generation
to data capture on storage devices.

4.3.2 PForDelta - Inverted Index Compression

PForDelta [81] (standing for Patched Frame-of-reference Delta) is a method for efficiently
compressing inverted indexes, and is frequently used in the context of indexing and search
over unstructured data, such as documents and webpages [79, 76]. As ALACRITY
presents a viable inverted index-based method for encoding scientific data, it presents a
perfect opportunity to further reduce storage overhead by applying PForDelta.
PForDelta encoding operates on a stream of sorted integer values, divided into fixed-size chunks. Each chunk is transformed to encode the first value (the frame of reference) and the differences between consecutive values. A fixed bit width $b$ is then selected to encode the majority of deltas – those remaining are stored as exceptions in a separate exception list. The majority of deltas typically require a far fewer number of bits to encode than the original data, and a relatively small chunk size (128 elements in the original work) enables a highly adaptive, per-chunk selection of $b$.

4.4 Method

4.4.1 Overview

Figure 4.1: Overview of the DIRAQ indexing methodology, with lightweight encoding$^{1,2}$, scalable index aggregation$^{8,9}$ and resource-aware dynamic aggregator selection$^6$.

Figure 4.1 illustrates the logical flow behind DIRAQ in-situ indexing. Our method consists of the following components:

1. **Storage-lightweight, query-optimized data encoding** that produces a compressed form of the data with an integrated index, combining both data and index compression. By achieving a low storage requirement while also supporting opti-
mized range queries, this encoding approach enables us to surmount the obstacle that the HPC I/O bottleneck represents to in-situ indexing. This component is explained in Section 4.4.2.

2. **Scalable, parallel data reorganization**, which reduces I/O time and improves post-simulation query performance by aggregating encoded data, without incurring unacceptable global communication costs. For comparison, existing indexing methods produce either a single, global index, or a highly-fragmented set of per-core indexes. Unfortunately, the former can only be accomplished with post-processing or expensive global communication, and the latter results in degraded query performance (as demonstrated in Section 4.5.2). In contrast, our group-level index aggregation technique largely avoids both of these drawbacks. We also leverage the hardware-level RMA support in modern HPC clusters to perform fast in-network merging. These aspects of DIRAQ are discussed in Section 4.4.3.

3. **Resource-aware dynamic aggregator selection**, which incorporates the inherent network topology and available memory constraints of a running system to improve the choice of aggregator cores at runtime. Leveraging this information helps to achieve improved performance over MPI-IO two-phase I/O [22]. More detail is given in Section 4.4.4.

### 4.4.2 Lightweight, Query-optimized Data Encoding

The first step for DIRAQ is to address the fundamental challenge of in-situ processing, and indexing in particular: because simulation I/O blocking time is already a serious concern today, any in-situ process must ensure this metric is not substantially impacted. However, while there exist parallel indexing methods for scientific data, no current method has addressed this issue sufficiently to operate within the context of a running simulation. The root of the problem is two-fold. First, current indexing methods inherently increase total storage footprint (data + index), commensurately increasing I/O times. Second, computation time for current indexing methods dominates even the I/O time.

We address this problem in DIRAQ by extending ALACRITY, a storage-lightweight indexing method with integrated data compression, by augmenting it with index compression to achieve even higher overall compression. By thus reducing both components
of the output, we reduce the storage footprint to 55%-90% (a compression ratio of 1.1-1.8x) for 9 variables from FLASH [25] simulation data (Table 4.1). Note that these compression ratios include both index and data; this implies the data encoding can simultaneously support efficient range queries while also reducing total simulation I/O, a win-win scenario.

As with ALACRITY, the DIRAQ query-optimized encoding format enables two-sided range queries of the form $a < var < b$ to be evaluated efficiently ($a$ or $b$ may also be $-\infty$ or $+\infty$, respectively, allowing one-sided range queries, as well). Given such a constraint, DIRAQ can retrieve the matching values (value retrieval) and/or the matching dataset positions (position retrieval). Additionally, DIRAQ supports an approximate query mode that accepts a bounded per-point error in order to perform an index-only query, which has benefits to query performance. These query types are discussed in more detail in a previous paper [33]. In pseudo-SQL syntax, the index supports queries of the forms:

- \[ \text{SELECT } var \ [\text{and/or}] \ position \ \text{WHERE} \ a < var < b \]
- \[ \text{SELECT } \sim var \ [\text{and/or}] \ position \ \text{WHERE} \ a < \sim var < b \]

An overview of the encoding format produced by DIRAQ is depicted in Figure 4.2. The process of binning is largely similar to that presented in the ALACRITY paper [33], as reviewed in Section 4.3.1, although some modifications have been made (including a generalization to support bit-level, rather than byte-level, adjustment of the high/low-order division point). However, the major extension that enables this encoding methodology to be used for in-situ indexing is the incorporation of index compression, and thus this component is described in detail next.

### Inverted Index Compression

In DIRAQ, an inverted index is used to map binned values to their (linearized) positions in the original dataset, referred as “record IDs,” or “RIDs.” Within each bin, we keep the low-order byte elements sorted by ascending RID, which enables us to apply PForDelta to the inverted index of each bin. Thus, the problem of index compression in DIRAQ can be reframed as compressing sorted lists of unique integers; the chosen method can then be applied to each bin’s inverted index independently to achieve overall index compression.
Instead of using general-purpose compression routines, we use a modified version of PForDelta to leverage the specific properties of our inverted index. As stated in Section 4.3.2, PForDelta operates on a sorted list of integers, which matches the nature of DIRAQ’s inverted indexes. Furthermore, PForDelta achieves high compression ratios for clustered elements; in DIRAQ, the inverted index RIDs typically exhibit good clustering, as these spatial identifiers are grouped by bin, and thus correspond to similar-valued points, which tend to cluster spatially.

We make modifications to PForDelta to achieve higher compression ratios for our application. The first difference is in the method for specifying the positions of exceptions in the delta list for patching. As opposed to the original approach, which uses a linked list of relative offsets embedded in the delta list, we instead leverage the fact that all RIDs are unique, implying that all deltas are strictly positive. This permits us to place 0’s in the delta list to mark exception positions, as they do not normally appear, thus eliminating the need for an embedded linked list and forced exceptions. These 0’s have low overhead, as they are bit-packed along with the deltas.

The second modification we implement is to deal with overhead when compressing small chunks. While our implementation maintains the fixed 128-element chunk sizes used in the original PForDelta work, the last chunk may have fewer elements. If the original input stream (inverted index bin, in our case) has far fewer elements than one chunk (128), PForDelta’s chunk metadata may become dominant, reducing compression ratios or even slightly inflating the index. This situation occurs for datasets with high variability (when modeling turbulence for example), which leads to a large number of small bins. To prevent this issue from unduly harming overall compression, we add a dynamic capability in PForDelta to selectively disable compression of chunks that are not successfully reduced in size.

**Effectiveness of Index Compression**

As a preliminary microbenchmark, we evaluate the overall throughput and compression ratios of DIRAQ’s encoding on 6 single-precision and 3 double-precision datasets from the FLASH [25] simulation. The test platform is the Intrepid BlueGene/P supercomputer. For each dataset, we take 256 compute cores worth of data, and encode each independently (simulating the core-local indexing that will be used in the next section), reporting the mean statistics in Table 4.1.
Compression Ratio: In our microbenchmarks, we observed index compression ratios from 3x to 20x across different variables and different refinement levels in the FLASH dataset. We observe that the compression ratio directly correlates with the number of bins in the dataset encoding, as shown in Table 4.1. With fewer bins, each bin’s inverted index contains a larger portion of the RIDS for the dataset. This yields a denser increasing sequence of RIDS per bin, and thus smaller delta values, which ultimately results in a high compression ratio due to PForDelta bitpacking with 1 or 2 bits per element. Even with less compressible datasets like vely, velz, we observe $\approx 2.5x$ compression of the index.

Overall, this level of compression results in substantial I/O reduction, and also improves aggregation performance (as discussed in Section 4.4.3). For comparison, when encoding these same datasets using the original ALACRITY method, total storage footprints (data + index) are all around 150% for the single-precision datasets, and 125% for those with double-precision. Over the datasets we evaluate, we observe effective I/O throughput to be increased by a factor of $1.5 - 2.5x$ based on this data encoding method.

Throughput: The indexing throughput is also dependent on the number of bin values, but to a lesser degree as compared to the compression ratio (max-over-min variation of about 1.4x vs. about 10x). This is primarily due to the use of fixed-size chunks during index compression, each of which is processed independently. However, since the compression is considerably faster than I/O access and has the net result of reducing the data footprint, the variation in indexing throughputs of DIRAQ do not contribute to a noticeable variation in end-to-end times, unlike with other compute-intensive indexing...
Table 4.1: Effect of dataset entropy (number of bins) on index compression ratio and total size on 256 processes each indexing 1 MB of data on BG/P.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average Bins</th>
<th>Index Compression Ratio</th>
<th>Total Size (% of Raw)</th>
<th>Encoding Throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flam</td>
<td>8</td>
<td>19.8</td>
<td>55</td>
<td>18.7</td>
</tr>
<tr>
<td>pres</td>
<td>1</td>
<td>20.6</td>
<td>54</td>
<td>18.7</td>
</tr>
<tr>
<td>temp</td>
<td>7</td>
<td>17.1</td>
<td>55</td>
<td>18.7</td>
</tr>
<tr>
<td>velx</td>
<td>339</td>
<td>4.3</td>
<td>78</td>
<td>17.6</td>
</tr>
<tr>
<td>vely</td>
<td>1927</td>
<td>2.4</td>
<td>90</td>
<td>14.0</td>
</tr>
<tr>
<td>velz</td>
<td>1852</td>
<td>2.5</td>
<td>89</td>
<td>14.1</td>
</tr>
<tr>
<td>accx*</td>
<td>172</td>
<td>3.3</td>
<td>88</td>
<td>15.1</td>
</tr>
<tr>
<td>accy*</td>
<td>166</td>
<td>3.3</td>
<td>88</td>
<td>15.0</td>
</tr>
<tr>
<td>accz*</td>
<td>176</td>
<td>3.3</td>
<td>88</td>
<td>15.0</td>
</tr>
</tbody>
</table>

* double-precision datasets.
+ total data written out as a % of original raw data.

techniques [17, 16].

4.4.3 Scalable, Parallel, In-situ Index Aggregation

In the simplest case, we can parallelize our indexing method by applying it to all compute cores simultaneously, generating core-local indexes, similar to FastQuery. However, this method of parallelization produces fragmented indexes, which are poor for query performance. Queries must necessarily process each core-local index in turn, incurring numerous disk seek costs and other overheads. Thus, it is desirable to build an index over aggregated data. However, performing global data aggregation is not scalable due to expensive global communication, so we instead consider group-level index aggregation, where compute nodes are partitioned into fixed-size groups, balancing the degree of index fragmentation (and thus query performance) with I/O blocking time (and thus simulation performance).

A particular challenge in building an index over a group of processes is the distribution of computation over the group. Forwarding per-core data to a small number of “aggregator cores” to be indexed poorly utilizes compute resources. Instead, we make the observation that core-local DIRAQ indexes can be merged efficiently. However, merging the indexes solely using aggregator cores again underutilizes the remaining compute resources.
cores, which sit idle during that time.

Given these challenges, we opt for a third option, shown in Figure 4.3, that eliminates compute bottlenecks at the aggregator cores. After each core in the group locally encodes its data, a group "leader" core collects the local index metadata and computes the group index metadata. The group index metadata is then used by each compute core to implicitly merge the indexes in-network by using Remote Memory Access (RMA), thereby materializing the group index fully formed across any number of chosen aggregator cores. Using this method, aggregators and all the other group cores perform the same degree of computation, as well as avoid per-core file writing, at the small cost of a metadata swap (which would be necessary, regardless).

Having given an overview of the aggregation method in DIRAQ, we now examine each step in more detail. Note that, while we focus on index aggregation this section, because DIRAQ’s encoding tightly integrates the index and low-order bytes, an equivalent aggregation and merging process is also applied to the low-order bytes. Thus, unless specifically noted, every step in the follow procedure is applied to both the index and low-order bytes simultaneously, but we discuss in terms of the index for brevity.

![Figure 4.3: A logical view of group index aggregation and writing.](image-url)
Building the group index

The DIRAQ encoding technique is especially suited for \textit{in-situ} index aggregation, as it enables efficient determination of the group index layout before any actual data or index is transferred. Recall that our encoding bins data according to values that share the same most significant bits (called the bin’s “header value”). Because this number of significant bits is fixed across all cores, if bins with the same header value appear on multiple cores, all these bins will have equivalent value boundaries, and will furthermore be part of the same bin in the group index, greatly simplifying the index merge operation. Furthermore, because the data within a bin is ordered by RID, and we assign a contiguous range of RIDs to each core, merging several local bins into a group bin is a simple matter of concatenation. Note that “bin” here refers to both the bin’s low-order bytes and its associated inverted index.

The process for building the group layout on the leader core is shown in Figure 4.4. First, the layouts for all core-local indexes within the group are collected to a “group leader” core. Next, the leader takes the union of all bin header values for the group, and then determines the offset of each local index bin within the corresponding group index bin, with bins from cores with lower RID ranges being filled first. Finally, the newly-constructed group layout is disseminated back to the compute cores, along with the set of bin offsets specific to each core (referred to as “bin placements”). Note that indexes could also be aggregated across groups, by exchanging bin metadata between group leaders. We expect to study this possibility in future work.

The end result is that each core obtains a complete view of the group index to build, as well as precise locations for its local index bins within that group index. In other words, the cores now collectively have a logical view of how the aggregation should proceed. It is important to note that this cooperative layout construction process constitutes only a small part of the overall end-to-end indexing time, and so we focus the bulk of our optimization efforts on the more time-intensive aggregation and write phases next.

Index aggregation via \textit{in-network} memory reorganization

After all cores in the group receive the group index layout and core-specific bin placements, the cores proceed to transfer their local bins to the appropriate locations within the group index. This mapping is straightforward at this point; the bin placements include an offset and length within the group index for each local bin. However, simply
writing at these offsets to a shared file will result in $N \cdot B$ qualitatively random write operations hitting the parallel file system servers at once (where $N$ is cores per group, and $B$ is average local index bins per core). With $N \approx 64$ and $B \approx 4,000+$, typical in our experiments with FLASH, the resultant 250,000+ I/O requests per group would be prohibitively expensive, and an alternative approach must be sought. Furthermore, the straightforward usage of MPI-IO would necessarily require numerous, small writes and high communication overhead, corresponding to the highly interleaved mapping between core-local and group indexes.

Our solution leverages a cooperative in-network memory reorganization technique to utilize the RMA capabilities of modern network infrastructure to perform index merging during network transfer. Using this method, the system can materialize the group index fully-formed across the memory of a subset of the compute cores in the group, called the “aggregator cores.” The process proceeds as follows:

1. A set of aggregator cores with sufficient memory to collectively accommodate the group index is selected. This is done by piggybacking available memory statistics from each core on the existing local layout messages sent to the group leader, and having the leader disseminate the selection along with the group index layout and bin placements. We examine the importance of selecting topologically-separated and memory-balanced aggregators in Section 4.4.4; for now, the specific selection
criteria are not pertinent.

2. After this selection, the memory commitment needed for the group index is balanced as evenly as possible across the aggregators, which then expose the required memory to all the compute cores for RMA operations.

3. Finally, all compute cores treat the exposed memory on the aggregators as a single, contiguous memory buffer, and use their bin placements and knowledge of the group index layout to inject their local index bins onto the proper aggregators at the right offsets. These data transfers utilize MPI one-sided RMA operations, using relaxed consistency and reduced-locking hints to achieve increased performance.

4. After RMA synchronization is completed, the group index is fully formed on the aggregators, which then simply write out the entire contents of their exposed memory window in sequential chunks on disk, completing the process.

By circumventing the need for an in-memory index merge, the computational load on the aggregators is reduced (which is important because there are few of them relative to the number of cores in the group). Furthermore, this approach also eliminates the need for temporary swap buffers required during an in-memory merge, and so is more robust in the face of limited spare memory available under many real-world simulations.

4.4.4 Optimizing index aggregation using memory- and topology-awareness

One of the most common techniques employed for I/O is two-phase MPI collective I/O, which performs a data aggregation phase prior to writing. However, the number and placement of aggregators within MPI, which can be tuned using "hints," does not take topology considerations into account, leading to network hotspots and other performance degradations [66]. Hence, recent works have explored topology-aware mapping for Blue-Gene/P [66] and tuning the aggregation ratio [66, 13].

However, these static techniques are not directly applicable within DIRAQ for the following reasons. First, the use of index compression results in varying data sizes written out by process groups. Second, with DIRAQ, the aggregation phase not only includes a simple data transfer, but also an in-network index merging strategy. Thus, the in-network
Algorithm 1: Topology-aware aggregator assignment on group leader processes

**Input**: \( n \): Number of processes in the group.

**Input**: \( M[1,...,n] \): Free memory size of each process in the group.

**Input**: \( d \): Average amount of data per-core.

**Input**: \( b \): Average number of bins per-core.

**Output**: \( a \): Estimated ideal number of aggregators.

**Output**: \( t \): Estimated neural-net time.

**Output**: \( R[1,...,a] \): Ranks of aggregators.

**Output**: \( O \): Start and end offsets associated with each aggregator.

\[
\begin{align*}
\text{minAggs} &= \text{getMinAggregators}(M,n,d) \\
\text{maxAggs} &= \text{getMaxAggregators}(M,n,d) \\
t &= \infty \\
\end{align*}
\]

4 // Estimate the optimal number of
5 // aggregators using trained model
6 for numAggs = \{minAggs, ..., maxAggs\} do
7     estimatedTime = NNEstimate(numAggs,d,b)
8     if \( t>\text{estimatedTime} \) then
9         \( t = \text{estimatedTime} \)
10        \( a = \text{numAggrs} \)
11     end
12 end

13 // Try placing aggregators using
14 // topology-aware settings
15 for aggSet \( \in \{\text{topologyAwareAggregatorSetList}\} \) do
16     if fit \( (a,aggSet,M) \) == \( TRUE \) then
17         \( R = \text{assign}(a,aggSet) \)
18         \( O = \text{generateOffsets}(a,R,d) \)
19         return \{a,t,R,O\}
20     end
21 end

22 // Generate a valid random placement
23 \( R = \text{generateRandomAggregators}(a,M) \)
24 \( O = \text{generateOffsets}(a,R,d) \)
25 return \{a,t,R,O\}

aggregation performance is based on a number of changing parameters, such as differing bin layouts across write phases, and so requires a more dynamic approach.
To account for these time-variant characteristics in DIRAQ, as well as the highly interleaved (core-local) I/O access patterns DIRAQ produces, data aggregation requires a strategy in which the number of aggregators/writers evolve according to simulation data characteristics. In DIRAQ, the group leaders are a natural fit for driving this process, as they have a complete view of the aggregated index and are responsible for distributing aggregator information to the other cores in the group.

We build an optimization framework that can dynamically select the number of aggregators, given the group index and low-order byte layout, while leveraging the work done in GLEAN [66] to control aggregator placement. Since this layout has numerous, interacting characteristics, our initial study found that rigid, linear models insufficiently captured the relationship between group layout and I/O. Hence, we train a neural network, bootstrapping offline to model and optimize DIRAQ aggregation parameters. Our choice of using neural-network-based learning is based on the fact that it is suitable to learn representations of the input parameters that capture the characteristics of the input distribution and subsequently carry out function approximation, especially when modeling non-linear functions [30, 77]. The topology- and memory-aware strategy is described in the Algorithm 1. The details of the performance modeling are explained in the following section.

Both the neural network and the list of topology-aware aggregator sets are determined simultaneously using a set of offline microbenchmarks. The usage of the neural network warrants further discussion (See Section 4.4.4). We estimate the execution time of the aggregation process as a function of the number of aggregators. Then, after pruning the possible number of aggregators based on memory constraints, we run the neural network over each possible number of aggregators, and choose the number that is predicted to have the minimal completion time. By estimating the completion time, the leader can perform error propagation based on the actual time taken. Furthermore, we observed negligible computational overhead (on the order of milliseconds), even when running multiple iterations of the neural network estimation.

**Performance Model**

The goal of the neural network-based performance model is to accurately predict the performance of three components: the index and low-order byte aggregation times as well as the I/O times for DIRAQ, both with and without inverted index compression.
Given these predictions, we can apply the model on each group to determine at runtime a well-performing set of aggregators. Furthermore, we focus on the BlueGene/P architecture, though our methods can be applied to other cluster architectures. Table 4.2 gives the necessary parameters.

Given the BlueGene/P architecture, the DIRAQ indexing framework consists of three components, namely the compute cores, the aggregators, and the BlueGene I/O pipeline, which consists of the I/O forwarding cores and a Myrinet switch which provides connectivity to file server cores of a cluster-wide file system [66]. We assume an aggregation group of size $\rho$. An aggregation group is defined as a logical group of $\rho$ compute cores (with one of them also as the group leader) and corresponding $a$ aggregator cores. Each aggregation group forms an MPI communicator in our implementation. We model the aggregation and I/O process taking place in a single aggregation group.

In order to build an accurate model, we must take into account the RMA contention at the aggregators. To do so, we ran a set of microbenchmarks to measure the aggregation and I/O times, for varying parameters $\rho$, $B$, $d$, and $a$ (refer to Table 4.2). Linear regression is not suitable for modeling the non-linear relationship $t_{agg,io} = f(\rho, d, b, a)$. Therefore, we used a 3-layered neural network with $\rho$, $B$, $d$, and $a$ as inputs, 40 neurons in the hidden layer and the $t_{agg,io}$ as the output. This is further used to determine the optimal number of aggregators in Algorithm 1.

We collected measurements for the aggregation and I/O times for various combinations of $\rho$, $d$, $B$, and $a$. We then trained the neural network using the FANN neural network library [49] with a total of 630 such samples. We used a 85 – 15% division into the training and testing subsets. With this configuration, we obtained a mean squared error of $1.15e^{-4}$ and an $R^2$ statistic of 0.9812 on the test data using the iRPROP training algorithm [31] and the symmetric sigmoid (tanh) activation function. On the contrary, a simple linear regression model resulted in a $R^2$ statistic of 0.553.

**Case 1 : Without compression**  In this case, the local layout of the index corresponding to the target variable is first built on every compute core. Then, the local index generation takes place which is followed by the process of building the global index layout and transferring the index metadata. Note that the former is purely a computation step, while the latter primarily involves communication between the compute cores. The index and low-order bytes are then aggregated followed by the initiation of the index and low-order bytes I/O operations. During I/O, the compute cores produce the data for the
Table 4.2: Parameters for the performance model.

<table>
<thead>
<tr>
<th>Fixed Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho )</td>
</tr>
<tr>
<td>( e )</td>
</tr>
<tr>
<td>( s )</td>
</tr>
<tr>
<td>( \gamma )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Run-time Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^* )</td>
</tr>
<tr>
<td>( d )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( l )</td>
</tr>
<tr>
<td>( \sigma )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bootstrapped Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu_w )</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{agg,io,index} )</td>
</tr>
<tr>
<td>( t_{agg,io,LOB} )</td>
</tr>
</tbody>
</table>

Iterated over by leader node for optimization.

next time step and initiate the corresponding local layout generation.

The index and low-order byte aggregation steps involve the compute cores writing at known offsets in the aggregator core’s memory via one-sided RMA calls. The indexing scheme reduces the data by about \( \gamma = \frac{s}{8 \cdot e} \) of its original size. Thus,

\[
t_{agg,io,index} = f(\rho, l, B, a)
\]

(4.1)

\[
t_{agg,io,LOB} = f(\rho, (1 - \gamma)d, B, a)
\]

(4.2)

**Case 2 : With compression**  This scenario includes an additional index compression stage. In this case, we add the index compression phase after the local index generation at every compute core. The low-order byte aggregation and I/O remains the same. However, the index aggregation and I/O take place over the compressed index. Thus,

\[
t_{agg,io,index} = f\left(\rho, \frac{l}{\sigma}, B, a\right)
\]

(4.3)

Using the above equations for index and low-order byte aggregation, and the values
of \( \rho, d, \) and \( B \) as input, we determine the optimal number of aggregators for both the index and low-order bytes as the one which results in the minimum \( t_{\text{agg}, \text{io}} \) times:

\[
\hat{a}_{\text{LOB}} = \arg\min_a t_{\text{agg}, \text{io}, \text{LOB}}(a) \quad (4.4)
\]
\[
\hat{a}_{\text{index}} = \arg\min_a t_{\text{agg}, \text{io}, \text{index}}(a) \quad (4.5)
\]

**Accuracy of Performance Model**

In this section, we analyse the accuracy of the trained neural network in two scenarios. One, where the model predicts the aggregation and I/O times when the aggregation ratio is fixed, and the other when model picks the aggregation ratio. This is done at the group leader nodes, which has the necessary parameters like the number of bins, compression ratios and access to the trained neural network to make smart decisions. Figure 4.5 shows the predicted accuracy on component timings when the aggregation ratio is fixed for 4 variables from FLASH simulation, with 1 MB of data indexed and compressed at each core.

**4.5 Experimental Evaluation**

With regards to our parallel, *in-situ* indexing methodology, there are three primary performance metrics we evaluate. The first, query performance, serves two purposes: it gives
a comparison between DIRAQ and FastQuery indexes, and it underscores the need for defragmented indexes to accelerate query-driven analytics. The second, index performance, evaluates our group-wise index aggregation methodology, looking at index size, generation speed, and scalability. The final metric, end-to-end time (which is broken down stage-by-stage for better insight), evaluates the effectiveness of our memory- and topology-aware optimizations, as based on dynamic neural network application, as well as our methodology as a whole.

4.5.1 Experimental Setup

Our experiments were conducted on the “Intrepid” system at the Argonne Leadership Computing Facility (ALCF). It is a 40K node quad-core BlueGene/P cluster consisting of 80 TB memory delivering a peak performance of 557 TFlops. Unless otherwise specified, evaluations were done on the General Purpose Filesystem (GPFS) [58] with the default striping parameters. Each experiment below was repeated 5 times and the reported numbers correspond to the median times for each of the results below. All experiments were performed in the “VN” mode in which one MPI process is launched on every core on the chosen set of nodes.

Before we describe the results, we would like to point out that the GPFS storage utilization was over 96%, at the time of our evaluation. As a result, the practically observed I/O throughput (Figure 4.9) is only a fraction of the theoretical maximum throughput.

We execute DIRAQ with the Sedov white-dwarf FLASH simulation [25] and analyze its in-situ performance on 9 different datasets (6 single-precision and 3 double precision). However, for the sake of brevity, we present the results on 4 datasets. Based on the dataset entropy (Table 4.1), the chosen single-precision variables temp, velx, vely can be considered representative samples of all the datasets used in this study. Since, the number of unique values (bins) in the index directly relates to the compression ratio we take datasets that have low (100–200), medium (1000–5000) and high (>10000) number of global bins, which correspond to temp, velx, and vely, respectively. Additionally, we choose one double precision dataset accx as well.

Additionally, to analyze the performance of index compression and querying across simulations, we show results on 4 datasets from S3D combustion simulation as well. We do not include scaling results from S3D in this work, since S3D does not run on Intrepid, and our optimizations related to topology-aware aggregations exploit BlueGene/P-specific
We analyze the serial query performance over S3D datasets on the Lens Analysis Cluster at Oak Ridge National Laboratory. Each node of Lens consists of four 2.23 GHz AMD Opteron quad-core processors and is equipped with 64 GB of RAM. We run the queries over the Lustre File System with the default striping parameters.

### 4.5.2 Query Performance

We first demonstrate the importance of aggregation for post-simulation query performance. Figure 4.6 depicts the serial query performance of DIRAQ and FastQuery with full-precision value retrieval (meaning exact values are returned). We use single-sided range queries of the form \( \text{var} < b \), where \( b \) is selected to induce a particular query selectivity. We perform this experiment using multiple partition sizes (that is, amounts of data indexed as a unit), ranging from 1 MB to 1024 MB on a 2 GB dataset, while fixing query selectivity (i.e., the fraction of data satisfying the query) at 0.1%. Note that FastQuery is constrained to a partition size equal to the amount of data available per variable per core, as the algorithm produces a local index for on core (though all such indexes are stored contiguously on disk). In contrast, DIRAQ can produce larger partition sizes by increasing its aggregation ratio, even when per-core data is low.

The figures show two trends. First, the DIRAQ indexing scheme, being lightweight
in both computation and storage, outperforms FastQuery’s method given a particular partition size. Second, for both methods, query performance is directly proportional to index partition size, presumably because the number of seeks induced is inversely proportional to the partition size, while the sizes of any contiguous reads are reduced. This performance characteristic of indexing in general is precisely our motivation for opting to perform index aggregation, rather than core-local indexing.

4.5.3 Indexing Performance

The performance of an indexing methodology can be considered in numerous contexts: a storage context, a computational context, and a scalability context. The following sections explore these contexts, providing a finer grain performance measure and analysis of individual tasks in DIRAQ.

Figure 4.7: Resulting index sizes (as a % of raw data) on varying amount of data aggregated per-core with FastQuery (FQ) and DIRAQ (CII) indexing techniques.

(a) Double-precision S3D simulation datasets.  (b) Single-precision FLASH simulation datasets.

Index Size

Figure 4.7 shows the index size generated for both single-precision FLASH variables as well as double-precision S3D variables over 256 MB of raw data. As with the query processing results, we experiment with multiple partition sizes to ascertain the effect of index fragmentation on index size. FastQuery’s index size, as a proportion of the raw data, slightly increases for smaller partition sizes, and the difference in resulting index
sizes using DIRAQ is near imperceptible. While the index metadata size increases in proportion to decreasing partition sizes, the overall effect is negligible. This means that, when considering the size of aggregation groups in DIRAQ, overall index size can be disregarded as an optimization parameter.

**Indexing Speed**

The core-local indexing throughput was shown in Table 4.1. In this section, we instead look at end-to-end indexing performance through scalability metrics, as well as study the stage-by-stage timing breakdown of DIRAQ with and without compression.

**Scalability**  It is crucial that DIRAQ exhibit favorable scalability properties, as it aims to index large-scale data. To this end, we have performed both strong and weak scalability benchmarks, simulating the indexing of one variable during one timestep. In both benchmarks, we keep the aggregator group size constant. In the strong scalability experiment, the overall data size is kept constant at 2 GB, whereas in the weak scalability experiment, the data size per core is kept constant at 1 MB per core.

The weak scaling results vary as shown in the Figure 4.9. We observe that the throughput for DIRAQ with compression increases almost linearly with the number of cores in the simulation for all the variables, due to increasing utilization of I/O resources. We compare DIRAQ with the baseline case of writing the raw simulation output using MPI-I/O and POSIX file-per-process approaches, both evaluated using the IOR benchmark. With
Figure 4.9: Weak scaling on the effective throughput (original data size / end-to-end indexing time) for 3 FLASH simulation datasets indexing 1 MB on each core on Intrepid. PFPP, MPI-IO performs raw data writes with no overhead of indexing.

DIRAQ we observe throughput gains of approximately 5x and 6x on the velx and temp variables respectively, and close to a 2x improvement for the least compressible vely variable. On the other hand, DIRAQ without compression yields a similar performance as these baseline approaches. Note that, without index compression, DIRAQ writes out 1.5x more than the original raw output, but performs comparably to the other write schemes, likely due to our topology-aware aggregation.

The strong scaling results vary as shown in the Figure 4.8. Under strong scaling, the amount of data indexed per core reduces at larger scales. While the index size generated by DIRAQ is proportional to the input data size (Figure 4.7), and so remains constant, the data movement stage now generates smaller message sizes per bin. This slightly increases the network contention at larger scales, but since the available I/O throughput increases as we request more cores, and this is the dominant component, we see improvements in the end-to-end times.

Performance Breakdown We breakdown the performance of DIRAQ by analyzing the execution time of each stage in the pipeline for the 4 datasets, namely temp, velx, vely, and accx, with and without compression. We componentize the stages of DIRAQ into Encoding, Group Layout, Aggregation, and I/O.

The Encoding stage gives the total time involved in encoding the core-local data, including the optional PForDelta compression of the inverted index. The Group Layout involves sending the core-local layout to the group leader, which constructs the global layout and assigns aggregators in charge of performing I/O. The Aggregation component includes the time taken to send both the index and low-order bytes to its corresponding
Figure 4.10: Cumulative time spent on each stage of DIRAQ for 4 datasets from FLASH simulation on Intrepid, with each process indexing 1 MB of data using an uncompressed index (II), and a compressed inverted index (CII).

The I/O component includes the time taken to perform the I/O operation on the aggregators, and the waiting-time on the non-aggregator processes.

Figure 4.10 shows the total time involved in creating aggregated indexes for the chosen datasets, with and without compression. Each process operates on 1 MB of data and a group, consisting of 256 processes uses 32 aggregators (writers). In almost all the cases, compression has a positive impact by reducing the amount of data moved within the network and I/O. However, the time spent in each stage of the pipeline varies with the entropy of the candidate dataset. For example, the time spent in the aggregation and group layout phases of the temp variable (highly compressible) is significantly less than the same for the vely variable (least compressible). We further discuss the breakdown results for the temp and vely variables since they are representative of the dataset entropy spectrum. We observe an average case performance for the velx and accx variables.

- **Encoding**: The encoding time at local cores is affected by the number of unique bin values present in the data. The dataset vely, which has 100x more unique bins than temp, requires around 2x more time to complete the encoding process. Even with the application of compression, which roughly doubles the amount of encoding time, this stage consumes less than 5% of the total execution time.

- **Group Layout**: Unlike the encoding process, which is local to each core, the group layout stage requires M:1 communication within each group for merging the bin layouts from each process. However, the group-leader performs more work when
the dataset has a larger number of unique bins, since calculating the offsets for each core in the group-aggregated layout requires \( M \)-way list-merge-like operations that induce loop branching and semi-random access. This explains why \textit{vely} has a pronounced \textit{Group Layout} stage compared to the other datasets. Activating index compression adds another 10% to the group layout time, since additional index compression metadata must be communicated and merged.

- **Aggregation:** This step essentially converts a set of distributed, non-contiguous, core-local views of the index/low-order bytes into a smaller set of aggregated, contiguous segments of the index/low-order bytes on the aggregators. To achieve this, each process would need to make \( B_{\text{local}} \) RMA calls, where \( B_{\text{local}} \) is the number of local bins at that process. For datasets with few unique bins, very few RMA calls are required to cluster the index and low-order bytes by bin, which occurs quickly on the 3D torus network.

- **I/O and end-to-end times:** The number of bins, and the clustering factor of values determine the final compression ratio. For example, the variable \textit{pres}, and to a lesser extent \textit{flam}, possesses little variation in the indexed values on a single core. These datasets have indexes that are compressed by as much 20x, and thus have \( \approx 22\% \) and 35% less data to write to disk when compared with \textit{velx} and \textit{vely}, respectively. When compared with using an uncompressed inverted index, the amount of data written is reduced by as much as 2.7x, leading to end-to-end completion times that are up to 2.2x faster.

### 4.5.4 Resource Awareness

Figure 4.11 shows the performance of the aggregator selection mechanism on the dataset \textit{vely} at 4096 cores. The resource-aware aggregation algorithm chooses a well-performing number of aggregators. On the hard-to-compress dataset \textit{vely}, for example, the neural network predicts within a group, 8 aggregators for the index aggregation, and 16 for the low-order bytes aggregation, as opposed to a single, fixed aggregation ratio. Compared with static aggregation strategies, this results in 10-25% improvement in average throughput when writing to disk. The additional benefit from this scheme is that the variation in end-to-end times across groups is reduced as well, thereby reducing the idle time on some groups waiting to synchronize after I/O.
Figure 4.11: Comparison of different aggregation strategies (number of compute to aggregator process) on I/O and aggregation timings, with process ranks sorted by aggregation times. 1 MB of the dataset velly is indexed by each of the 4096 processes.

The neural net is inclined to pick a higher number of aggregators when indexes are less compressible. For other variables such as temp, which are highly compressible, an aggregation ratio of 64:1 (4 aggregators) enables aggregator writers to avoid making very small I/O requests to the filesystem. Because of topology-aware aggregator placement along with an aggressive compression scheme, aggregation times generally do not present a bottleneck when compared with I/O times.

4.6 Conclusion

This chapter describes DIRAQ, an effective parallel, in-situ method for compressing and indexing scientific data for analysis purposes during simulation runtime. DIRAQ produces a compressed index that is significantly smaller than state-of-the-art indexes. The combination of index compression and data reduction results in an encoding that, in many cases, is actually smaller than the original, unindexed data. By using a high-throughput local indexing and compression scheme followed by an effective in-network index merging and aggregation strategy, DIRAQ is able to generate group-level indexes with minimal overhead to the simulation. For our application, a custom aggregation scheme, along with an adaptive approach to choosing aggregator ratios, results in better performance compared to MPI collective I/O routines. Overall, DIRAQ presents an analysis-efficient data encoding that is smaller than the raw data in majority of the cases, offers faster query processing time than current indexing schemes, and can be generated in-situ with little-to-no overhead (and possibly an I/O performance improvement) for
simulation applications.
Chapter 5

Conclusion

In response to the challenging needs of effective scientific data management, real-time processing and scalable analytics, this thesis has proposed end-to-end solutions by utilizing \textit{in-situ} processing. The core theme behind all the techniques developed here is an aggressive data reduction strategy to offset the discrepancy between compute capabilities and I/O bandwidth. The first proposed approach was a novel compression technique named ISABELA [54]. ISABELA applies an effective data pre-conditioning strategy to transform data to a form that allows lossy, yet accurate approximation. A query-processing technique named ISABELA-QA [41] was subsequently developed on ISABELA-compressed datasets to efficiently perform range queries on “knowledge priors.” And finally, we developed a parallel \textit{in-situ} indexing technique (DIRAQ [40]) to build aggregated indexes without overhead to the simulation such that the data is analysis-ready as it leaves the compute nodes.

The fundamentally new compression and indexing techniques proposed in this work provide a scalable solution for query-driven analytics that is driven by multi-fold data reduction. These techniques significantly accelerate response times on range-queries, while imposing negligible overhead to scientific simulations. Thus, they validate the hypothesis of this dissertation.

5.1 Future Work

With increasing data sizes and complex requirements of analytics routines, there are several avenues where improvements can be made. This section briefly outlines some of
the future technical challenges and discusses possible approaches, based on the lessons learned in this thesis.

**Information-aware Data Encoding:** The underlying assumption behind this work is the use of compression and indexing across all processes before performing writes. However, not all data are created equal, where some parts of the data are bound to be more interesting than others. Thus, different parts of the data can be encoded differently according to the needs of the analysis routines. For example, the field data from simulations can be compressed using a lossy compression for analytics and visualization, whereas the particle data, which is used for checkpoint and restart, must be compressed using lossless compression methods. Additionally, indexing and compression could be judiciously applied by creating indexes on the interesting parts of the data, while aggressively compressing the interesting parts. This can lead better performance during both reads and writes. Further optimizations can be made during reads, where parts of the data that are accessed frequently can be refined further for multi-resolution analysis, while being cached or replicated, for faster query performance.

The primary research aspect of dynamic data encoding will come from the ability to identify the right compact representation for the right part of the data. For example, if the data is fairly smooth, functions such as splines and wavelets can be effectively applied or if the data happens to be sorted, approximations can be enhanced to take advantage to model monotonically increasing values. To do so, approaches to determine entropic content and data characteristics are necessary, and machine learning techniques or even coarse-grained analyses, might be an effective strategy in most cases. These techniques can be coupled with data partitioning strategies that take into account data pre-conditioning (transformations, sorting, etc.) and encoding strategies, so that different parts can be mapped to their suited encoding techniques. For example, rather than employing fixed size chunking as seen in ISABELA, adaptive windowing and partitioning strategies based on change-point and singularity detection can be used to partition the data into windows that have little variation. Alternatively, incompressible regions can be identified and written directly to disk without additional processing via asynchronous writing strategies.

**Spatio-temporal Indexing:** One possible avenue for further research that is of high interest to domain scientists is performing in-situ analytics across spatial and temporal contexts. There exist fundamental differences in access patterns, with data generation
being inherently distributed and core-local, compared to analysis requiring global spatial and temporal accesses. Building indexes across aggregated contexts during simulation can help solve this problem. However, it remains a challenging task due to limited available memory during simulation running. Even though the number of cores per processor is increasing, the amount of memory-per-core is actually decreasing. Thus, a reduced index footprint will be essential and indexing techniques must take full advantage of spatial and temporal properties in scientific datasets.

In Chapter 4 we extended indexing from a single process to a group of processes, thus covering a larger spatial context in DIRAQ. This was made possible because of the light-weight memory-efficient indexes that utilized the spatial characteristics in the data. And in Chapter 2, temporal properties were utilized to create “time-evolving” (differential) indexes, which were used to further compression rates in ISABELA. Combining both techniques can possibly reduce index sizes even further, making \textit{in-situ} query-processing on across a larger spatial and temporal context viable. The challenges here will lie in effectively utilizing the limited available memory per core during writes, and limiting the extra decoding of the differential indexes during query-processing.

Beyond differencing, the question remains whether any other form of temporal properties can be utilized. One would think that \textit{knowledge-priors} can be effective in characterizing the evolution of data across time and be used to create approximate indexes during real-time. However, the drawback of this approach is that domain knowledge would require considerable input from application scientist not only for every simulation, but also for different simulation configurations. The alternative approach would be a self-learning strategy with some input from prior runs. Though interesting, it remains to be seen how “meaningful” correlations can be identified in real-time to reduce index sizes without much overhead.

\textbf{Querying over Derived Data:} While some analytics are based on the raw simulation data, often times, queries are posed on data that is transformed into exponential or Fourier space. Clearly, creating indexes for all possible transformations is not a feasible approach. However, with inputs from domain scientists, we can identify the most commonly used transformation classes beforehand, and help accelerate queries in the transformed space. This can be done either by identifying a mapping from the index created in the original space, or by creating a new index in the transformed space on-the-fly. For example, consider the case where the user is interested in identifying points where
the magnitude of velocity \(|vel| = \sqrt{velx^2 + vely^2 + velz^2}\) is less than a certain threshold. We could use the index on \(velx, vely,\) and \(velz\) to prune the search space, since indexes are clustered by values, or alternatively create a new index on \(|vel|\).

**Spatial Indexing and Querying over Unstructured Datasets:** For unstructured datasets (for example, particle-data), spatial information is explicitly recorded with each element (particle). On these datasets, spatial queries are often used in conjunction with range queries. An example of one such query would be to retrieve the density of particles within a given region in space. To efficiently accelerate such queries, composite indexing of spatial values across *multiple variables* (poloidal, toroidal, radial values for each particle) become necessary. And, creating these spatial indexes on unstructured datasets, in parallel, *in-situ*, while taking advantage of effective compression, presents an interesting avenue for future research.
REFERENCES


99


