ABSTRACT

BOYUKA II, DAVID ANDREW. The Hyperdyadic Tree Index and a Generalized Parallel, In Situ Indexing Methodology for Extreme-scale Scientific Data. (Under the direction of Nagiza F. Samatova.)

Modern science increasingly relies on large-scale experimental and simulation data to develop new insights. As this data continues to grow exponentially, and as exploratory analytics and visualization become commensurately more challenging, many scientists turn to indexing and query process in their quest to continue identifying trends and anomalies.

In the realm of scientific data, indexes commonplace for transactional RDBMS workloads (such as the B+ tree) are often eschewed for those more suitable to data warehousing, in particular the (compressed) bitmap index. In this context, the bitmap index has been shown to outperform the B+ tree in both space and time. Thus, it is not surprising that implementations such as FastBit and FastQuery have been used in support of a number of scientific studies.

The recent development of the ALACRITY index has broadened these horizons, however. Using a completely different data structure, the delta-compressed inverted list, ALACRITY exhibits substantially smaller index sizes than those of compressed bitmaps for scientific data.

Yet, despite notables differences, ALACRITY and FastBit bear marked similarities, as well. These parallels indicate that the bitmap index may be just a single point, and ALACRITY another, in a larger design space that has yet to be fully explored. This observation sparks two central research questions that motivate this dissertation: first, "Can we discover other, related indexing methods with promising performance characteristics?"; and second, "Can we generalize the bitmap, ALACRITY, and these other related indexes under a single model encompassing all?" In this work, we answer both questions in the affirmative based on theoretical and empirical results.

To the former question, we present the Hyperdyadic Tree (or HD-tree) index. Based on a quadtree encoding method, this new approach requires substantially less storage than bitmap indexing (by 1.14-1.90x in our experiments on real-world scientific data) while still achieving superior or similar query performance in most cases, and while also maintaining key algorithmic characteristics of the bitmap index.

To the latter question, we present a formal abstract model defining the "value-sliced index," along with associated indexing and query algorithms. We then develop this model into a generalized indexing and query framework, PIQUE, unifying many prior indexing methodologies in one platform. We believe this system can facilitate both the development of new indexing research and the deployment of indexing and query technology to end users.

Finally, to effectively deliver our work to end users, we consider the related issue of integrating *in situ* indexing into a data pipeline. Several obstacles limit real-world adoption of in situ services, especially cost of code integration and lack of interoperability across the data pipeline. In response,

we demonstrate how indexing, as well as other “data transformations” (including compression, level-of-detail encoding, etc.), may be incorporated at the I/O middleware layer to overcome these challenges.

The Hyperdyadic Tree Index and a Generalized Parallel, In Situ Indexing Methodology for Extreme-scale Scientific Data

by
David Andrew Boyuka II

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

2016

APPROVED BY:

Rada Y. Chirkova	Kemafor Ogan
R. Raju Vatsavai	Nagiza F. Samatova Chair of Advisory Committee

DEDICATION

To my Lord and Savior Jesus Christ, from whom comes my strength and purpose.

Here I raise my Ebenezer; hither by Thy help I'm come.
— Robert Robinson, *Come Thou Fount of Every Blessing*

BIOGRAPHY

David (Drew) Boyuka began pursuing his Doctor of Philosophy in Computer Science in spring of 2012 at North Carolina State University under the direction of Dr. Nagiza Samatova and as a National Physical Science Consortium Fellow. Prior to this, he graduated with his Bachelor of Science in Computer Science in 2010 as a Park Scholar and valedictorian, and completed his Master of Computer Science degree in 2011, both at NC State. Along the way, he completed research internships at Microsoft Research Asia in Beijing, China (2010); Oak Ridge National Lab (2012); and the Department of Defense (2013).

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Nagiza Samatova. Her energy and focus in research have been an inspiration to me throughout my time in her group. Every graduate student relies on their advisor for guidance, but I believe Dr. Samatova has gone above and beyond in this regard. She has invested a tremendous amount of time in me, especially when I most needed it as a new PhD student. It is her leadership also that assembled such a high-caliber research group, which has been the catalyst for much of my intellectual growth here at NCSU. Finally, she always places her students first, being constantly on the lookout for ways to promote and expose us to leading researchers in our field. It is certain that her guidance has enabled me to accomplish much more than I could have thought possible, and without her I could not have reached this point.

Within the Computer Science department at NC State, several other faculty have had marked impact on my progress. I am grateful to the members of my committee, Rada Chirkova, Kemafor Ogan, and R. Raju Vatsavai, for their time and constructive criticism of this dissertation. From my time as a master's student, my co-advisor Xiaosong Ma gave me the grounding in research skills that has enabled me to succeed in the PhD program. I am especially grateful to George Rouskas for his role as my other master's co-advisor, an undergraduate mentor, and the teacher of some of the best courses I have taken at NCSU (his course on queueing theory and Markov processes in particular is directly responsible for my ability to complete much of Chapter 4). Similarly, the tutelage of Carla Savage, spanning both undergraduate and graduate courses, has been instrumental in developing my mathematical and algorithmic reasoning skills, which have been especially crucial in completing this dissertation. During my undergraduate studies, Robert Fornaro involved me in his research in wireless sensor networks, my first exposure to research work, and also mentored me throughout my time in the program; his student Douglas McClusky was also influential to me. Carolyn Miller was a wonderful teacher and mentor to me, and helped set my academic trajectory.

Several researchers outside of NC State have had marked impact on my studies. Scott Klasky brought me on as a summer intern at Oak Ridge National Lab, which formed the basis for my first paper; he is also responsible for securing the Jaguar/Titan supercomputer and Sith analysis cluster resources that made much of this dissertation possible. I also want to thank Qing "Gary" Liu and Norbert Podhorszki at ORNL for their collaboration as the developers and caretakers of the ADIOS project. Special thanks to Kesheng "John" Wu of Lawrence Berkeley National Lab for being an absolutely upstanding scientist who, despite working in a somewhat-competing area, freely provided access to both computing resources and valuable data on the Edison supercomputer at NERSC, and who has also offered kind and honest feedback on my research.

I am very thankful to my fellow students, as well, for their support, collaboration, and everything that they've taught me. I am especially grateful to Sriram Lakshminarasimhan for his steadfast

dedication to integrity in research, for many entertaining and enriching discussions, and for his significant contributions to many of my papers, especially the DIRAQ and ADIOS Transforms projects. Thanks as well to Jonathan Jenkins for being our group's "writing guru" and generally fun guy, and to Eric R. Schendel for his sage advise and expertise in code optimization. A contemporary of mine in the group, Xiaocheng "Chris" Zou has been a long-time collaborator, and good friend to me. There are many others I would like to thank for their collaboration and friendship: Houjun Tang, Zhenhuan Gong, Isha Arkatkar, Saurabh Pendse, Steven Harenburg, Stephen Ranshous, Gonzalo Bello Lander, and Kushal Bansal.

I owe much to those who mentored me before coming to the university. Foremost among these is Harold Reiter, a professor of mathematics at the University of North Carolina at Charlotte. Starting from early middle school, he involved me in mathematics competitions and invited me to the Charlotte Math Club he founded. Later, he offered me a scholarship to take a math course at UNCC, and hired me as his teacher's assistant three times for summer graduate courses in discrete mathematics taught for high school teachers. I could not have achieved all that I have today if not for his kindness and guidance. Susan Shaeffer and Jean Schoenheit volunteered their time to run a local "MathCounts" program, where my love of mathematics was first sparked under their teaching.

Most of all, I am thankful to my family for their unwavering support. My father David, mother Tina, brother Adam, and sister Caroline have been a blessing in my life in more ways than I can count. My mother deserves special thanks for the countless hours she spent home schooling me, which gave me the freedom to grow and thrive academically.

This work was supported by grants from the Department of Energy, Office of Science (SciDAC SDAV Institute) and the National Science Foundation (EAGER program).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Many fields of science are coming to rely on large-scale experimental and simulation data to an increasingly greater extent [53, 102]. As this data continues to grow exponentially in size, deriving new insights through exploratory analytics and visualization is becoming more challenging. For simulation data, this issue is compounded by the fact that analysis clusters are generally performance-limited relative to the supercomputers generating the data.

In response, many scientists turn to indexing and query processing to facilitate fast, interactive data probing and drilldown in their quest to continue identifying trends and anomalies within ever-expanding results. In some cases, indexing is truly indispensable in achieving tractable exploratory analysis and query-driven visualization, opening the door to insights into scientific data that are not otherwise possible [14]. Thus, indexing and query methodologies for scientific data continue to be an important area of continuing research.

In the realm of scientific data, indexing methods traditionally favored by transactional RDBMSs (such as the B+ tree [24] and its variants) are often eschewed in favor of those more suitable in a data warehouse context [80], in particular the *(compressed) bitmap index* [4, 61, 90]. In direct comparison, the bitmap index has been shown to outperform the B+ tree in both disk space and query time in a scientific data context [92]. A key reason for this is that scientific query workloads often involve constraints on many different variables, and sometimes on many at once in a single query. In this scenario, the bitmap index shines, as all variables may be individually indexed using this method, and these indexes are easily combined in multi-variable queries [81]. Another reason is that scientific data is generally *immutable*, removing from consideration the key disadvantage of expensive updates in compressed bitmap indexes. That scientific datasets are generally highly *structured*, being most naturally represented by arrays of values, also points to the use of bitmaps (arrays of bits) as a sensible approach. Thus, it is not surprising that implementations such as

FastBit [88, 91] and FastQuery [23] have been used in support of a number of scientific studies [14, 25, 75, 78, 97].

Recently, however, the ALACRITY [41, 42] system was presented as a viable alternative indexing paradigm for scientific data. Using a completely different data structure, the delta-compressed inverted index as adapted from the fields of web search and information retrieval [105], ALACRITY exhibits substantially smaller index sizes vs. compressed bitmaps.

Yet, despite these key design differences, ALACRITY and FastBit bear marked similarities, as well. To name a few: both systems associate a data structure (inverted list or bitmap) with each unique value in a dataset, both respond to queries by merging these data structures, and both employ a form of "binning" to handle high cardinality data (i.e., data with many unique values). Very recent work proposing the "Roaring" compressed bitmap [15], an approach combining both inverted lists and bitmaps in one data structure, underscores this insight.

These parallels indicate that the bitmap index may be just a single point, and ALACRITY another, in a larger design space that has yet to be fully explored. This observation sparks the central research questions that motivate this dissertation: first, "Can we discover other, related indexing methods with promising performance characteristics?"; and second, "Can we generalize the bitmap, ALACRITY, and any other related indexes under a single model encompassing all?" In this work, we answer both questions in the affirmative with both theoretical and empirical results.

To the former question, we present the Hyperdyadic Tree (or HD-tree) index, which is fundamentally different from both the bitmap and ALACRITY index approaches. This approach exhibits reduced storage requirements relative to bitmap indexing while also maintaining some nice algorithmic characteristics of bitmap indexes that are absent in ALACRITY.

To the latter question, we then present a formal abstract model defining the "value-sliced index," along with associated indexing and query algorithms. We then further develop this model into a concrete implementation of a generalized indexing and query framework. Not only does this framework unify many prior indexing methodologies, but we also believe it can facilitate the development of new research in this area, as well as the deployment of both present and future indexing and query techniques to end users.

Finally, to effectively deliver this work to end users, we consider the related issue of integrating indexing *in situ* into an data pipeline. Several obstacles limit real-world adoption, especially cost of code integration and lack of interoperability across the data pipeline. In response, we demonstrate how indexing, as well as "data transformations" in general (including compression, level-of-detail encoding, etc.), may be incorporated at the I/O middleware layer to overcome these challenges.

1.1 Thesis Statement

Following this background, we give the central thesis of this dissertation:

> *For scientific data, a quadtree-based indexing approach can yield comparable or superior query performance and a reduced storage footprint relative to the compressed bitmap index. Furthermore, both the bitmap index and this new method stand within broad design space of "value-sliced indexes," which can provide taxonomy for prior and future research in this space and can inform the design of a generalized parallel, in situ indexing and query framework.*

This statement can be broken down into three interrelated items, which correspond to the major chapters of this dissertation, and which are explained in more detail next:

1. the description and evaluation of our new "Hyperdyadic Tree" index (chapter 2), supported by theoretical results (chapter 4);
2. a formal model of the design space encompassing a class of indexing techniques, which covers both prior work and the new HD-tree index, along with a concrete implementation of these concepts called PIQUE (chapter 3); and
3. the development of a holistic in situ approach to indexing and other "data transformations" addressing obstacles to real-world adoption (chapter 5).

1.2 The Hyperdyadic Tree Index

To reiterate, the compressed bitmap index has some key advantages that make it ideal for use with scientific data. Of primary note are support for efficient multi-variable queries, and a reduced storage footprint versus traditional database indexes such as the B+ tree [92]. Additionally, some bitmap compression techniques, such as WAH [90] and its variants, have the desirable property of supporting bitwise operations (AND, OR, NOT) *without decompression*, saving both compute time and memory during query processing.

Nevertheless, compressed bitmap indexes still have a substantial storage footprint, often on the order of the size of the data itself [47, 92]; this remains true even with bitmap compression and despite being significantly smaller than equivalent B+ trees. As scientific data and computational power continue to increase in scale, we posit that use of storage-heavy indexes is becoming untenable, and that lighter-weight alternatives must be sought.

The newer ALACRITY index is one such storage-lightweight approach. However, it lacks the beneficial property of the bitmap index that query processing can proceed without decompression. Though an efficient decompression method exists to ameliorate the CPU cost of this shortcoming [104], it usually does so at the cost of higher memory requirements.

Ideally, we would like a storage-lightweight method that admits no-decompression query processing with similar performance to the bitmap index. This is the key motivation for our new indexing approach, the Hyperdyadic Tree index.

In seeking a more adaptive compression method than block-wise WAH and related approaches, we explore the use of the *quadtree* data structure. Specifically, we adapt the "CBLQ" image compression technique [51] from the field of computer vision. The key properties of this data structure, good compression and support for set operations without decompression, commend it to our consideration. After some substantial modifications and extensions to the core algorithms for manipulating CBLQs, we dub the altered data structure the "Hyperdyadic Tree," and demonstrate how it may be used for indexing.

We demonstrate the key benefits of the HD-tree index over a state-of-the-art compressed bitmap index. First, the HD-tree index is shown to be substantially smaller than the WAH compressed bitmap index, by 1.14x to 1.90x in our empirical testing under a range of conditions; these results are also supported by theoretical modeling of both HD-trees and WAH-compressed bitmaps. Second, query performance is shown to be superior or comparable to that of WAH compressed bitmap indexes in many typical cases.

1.2.1 Theoretical HD-tree Analysis

Having developed the core algorithms for HD-tree indexing and query processing and demonstrated promising real-world performance, the next logical step is to then analyze the properties of HD-trees from a theoretical perspective. This yields further insight into the types of scenarios in which HD-trees may exhibit superior performance.

In prior work studying expected WAH-compressed bitmap index sizes [93], two data models were proposed: the Bernoulli process (i.e. coin flipping), and a more advanced first-order Markov process. We build on this work, deriving analogous expected index size results for HD-trees under these models (though we also slightly modify the Markov process model itself).

We leverage both our HD-tree model and the existing model for WAH-compressed bitmaps (modified to be directly comparable), comparing these two indexing approaches under a variety of scenarios. In particular, we compare on a range of data cardinalities, clustering factors, and distributions/skews. We determine that the HD-tree index performs favorably relative to WAH for

mid-range cardinality datasets (about tens to thousands of bins), and for datasets with skew; both of these scenarios are common, and thus the HD-tree index is a useful alternative approach.

1.3 Generalizing Value-sliced Indexing

Having presented an alternative indexing method, the HD-tree index, we update and revisit our second research question: "Can we generalize the bitmap, ALACRITY, and HD-tree indexes under a single model encompassing all?" Though each of these indexing techniques is markedly different, closer inspection leads us to a clear decomposition of the indexing and query processes.

Specifically, our model defines value-sliced indexing as the composition of three distinct parts:

1. Quantization (i.e. binning): an initial data cardinality reduction via a mapping function with particular properties
2. RSet representation the basic data structure underpinning the index (e.g. bitmap, inverted list, HD-tree), governed by several constraints including required support for certain operations
3. Index encoding: different index composition strategies, previously developed solely for bitmap indexes, which enable trade-offs among index size, query I/O cost, and query compute cost

We formally define each component, describing how it fits within an overall generalized indexing algorithm. Also, we describe how such a generalized index may be queried under both single- and multi-variable constraints.

The development of this model has several specific benefits. For one, we believe that clearly defining the requirements for each index component will facilitate new research into yet more alternative indexing and query methods. As well, the unified model opens up possibilities for combining techniques originally designed for different types of indexes together in one index; for example, what were previously developed as "bitmap encodings" are now shown to be equally useful when applied to the HD-tree index. Lastly, this abstract model can be realized in a concrete software implementation, which in turn yields several other benefits as described next.

1.3.1 Parallel Indexing and Query Unified Engine (PIQUE)

Componentizing the indexing process enables the concrete software implementation of a unified indexing and query engine. We have developed such an implementation, the Parallel Indexing and Query Unified Engine (PIQUE). PIQUE is easily extensible, and functions as a platform on which to deploy a wide range indexing and query research, including the bitmap, ALACRITY, and

HD-tree indexes, to end users. As of now, PIQUE incorporates the following indexing techniques drawn from the research literature, with the addition of new such algorithms/data structures being straightforward:

1. Quantization: fixed bins, decimal-precision binning (FastBit) [68], significant-bits binning (ALACRITY) [42]

2. RSet representation: bitmaps, WAH-compressed bitmaps (FastBit) [93], inverted lists [41], PFOR-Delta compressed inverted lists (ALACRITY) [42], HD-trees (this paper).

3. Index encoding: equality (i.e. no encoding) [61], range [16], interval [17], binary component [16, 86].

On the basis of our abstract model, PIQUE factors out common indexing and query algorithms into its core. For example, on the indexing side, the code for large-scale parallelism, data loading, bin partitioning, and index I/O are all abstracted in a single algorithm, with minimal calls to the quantization, RSet representation, and index encoding plugins as needed. On the query side, query parsing, basic query optimization, index encoding inversion, and RID list conversion for final results are implemented once in the query engine core in terms of abstract plugin functionality. This both greatly reduces the effort needed to develop new plugins, and also ensures a "level playing field" when comparing different indexing approaches within the framework.

We evaluate PIQUE's indexing scalability on a large scientific dataset (VPIC [83, 84]) up to 4,096 cores on the Edison supercomputer at NERSC [56]. We also use PIQUE as the basis for comparing WAH-compressed bitmaps to HD-trees in indexing time, storage space, and query performance, simultaneously evaluating the performance of the HD-tree index as well as demonstrating the efficacy of PIQUE itself.

1.4 Transparent In Situ Data Transforms

Beyond the method of indexing used, the integration of indexing within the larger data analytics pipeline is another, related issue in the exploratory analysis and visualization of scientific data. Indexing is a member of a broader class of techniques we refer to as "data transformations," which also includes, e.g., compression [3, 13, 47, 52, 73], level-of-detail encoding [40, 62], and storage layout optimization [32]. As data scales increase, there is a move to apply data transforms *in situ* by co-locating with the application on compute cores, or *in transit* by utilizing dedicated staging nodes/cores, thus moving away from data *post*-processing to data *co*-processing. We assert that, in our discussion about scientific data indexing, it is important to explicitly address this shift.

Yet, despite offering numerous potential benefits, actual adoption of in situ data transforms by scientific applications has been inhibited by several obstacles. First and foremost among these is the yet-unresolved key question of *where to place data transformation services within the end-to-end scientific workflow*. The consensus, especially among end users, seems to be emerging in favor placement at the level of the *I/O middleware*, software that facilitates efficient I/O and storage of structured data.

In response to this critical need, we aim to bridge the growing gap between the increasing number of data transformation methods, including indexing, and the limited rate of their transparent integration within scientific workflows. To do this, we design a generic framework within the ADIOS I/O middleware for transparent, in situ data transformations.

By carefully designing this framework, we are able to reduce the cost of integration of data transformation services within scientific applications by maintaining *user transparency*, that is, keeping read and write semantics unchanged for the user. In particular, we avoid *data semantics erasure*, i.e., the degeneration of structured data into simple byte arrays when transforms are applied, as would be observed if compression, etc. were applied naïvely outside of the I/O middleware.

Additionally, our approach enables ease of transform experimentation, tuning, and swapping through *runtime configuration of data transform plugins*. Data transforms may be enabled, disabled, and reconfigured without code recompilation. Furthermore, we support I/O pipeline flexibility by explicitly *decoupling data transformations and I/O transports* in ADIOS, thus achieving orthogonality between these services. This permits users to continue using existing I/O transports while augmenting their pipeline with data transforms.

Finally, we are the first to support read-optimizing transforms in the I/O middleware. By incorporating a *flexible read model* into our transforms framework, the Transforms framework is able to convey the I/O performance benefits of transforms like level-of-detail encoding or space-filling curve reorganization transparently to the user. This is especially critical for indexing, the very basis of which is to read only a small portion of the index to answer a given query.

Using this framework, we evaluate the general benefits and drawbacks of data transformations for both writes and reads. On the write side, we apply multiple data transformations to the existing Quantum Lattice Gas [83, 84] (QLG) simulator on up to 8,192 cores on the leadership-class Titan supercomputer at Oak Ridge National Lab (without modifying the application code), demonstrating the framework's scalability, as well as the potential for in situ compression to reduce I/O time and storage footprint. On the read side, we analyze the performance of different read access patterns over transformed data. We show the impact of "transform opacity" (i.e., the atomicity of transformed data under subset read operations), and how our flexible read model enables some transforms to overcome this effect to improve I/O time, even outperforming the no-transform case.

Chapter 2

The Hyperdyadic Tree Index

2.1 Introduction

In the quest to identify trends and anomalies within an unprecedented amount of simulation and experimental data, many scientists turn to indexing and query processing. Indexing is often indispensable in making large-scale data exploration and query-driven visualization tractable, especially since most analysis clusters are relatively performance-limited compared to simulation supercomputers.

Many scientific datasets are structured, spatio-temporal, include many variables (i.e. columns), are read-/append-only, and/or are explored with highly multi-variate queries. In such cases, compressed bitmap indexing [14, 23, 61, 82] is widely used. Through evaluations with, e.g., the popular FastBit [91] implementation, bitmap indexing has been shown to outperform traditional database indexing for such scientific data in both space and time [92].

The key advantage of bitmap indexing in this context is its ability to efficiently combine query constraints over many variables into a single result. This is achieved by building a separate bitmap index on each variable of potential interest. Then, when a multi-variate query arrives, the individual query constraints may be evaluated on their respective indexes, producing a series of bitmaps each representing those records matching a particular constraint. These bitmaps may be efficiently combined using bit-wise operations corresponding to logical operators in the multi-variate query (such as *AND*, *OR*, *NOT*, etc.).

Building on this advantage, a further benefit offered by certain bitmap compression methods used in bitmap indexing (such as WAH [93]) is that these bit-wise operations can be applied *without decompressing the bitmaps at any point*. Instead, special algorithms are used to combine compressed bitmaps directly to produce another compressed bitmap. Such an approach can yield

significant savings in memory, and possibly compute time, especially for highly-selective queries (which generally involve highly-compressed bitmaps).

Together, these advantages highlight why (compressed) bitmap indexes are commonly used for scientific data, where highly multi-variate queries are common and very large datasets can benefit from low memory overhead during query processing. Yet, the bitmap index often has a heavy storage footprint, often of size equal to or greater than the original scientific data being indexed, even after compression and despite typically being smaller than an equivalent B+ tree index [92]. This is becoming a serious drawback in light of the exponential growth of extreme-scale scientific data, and so an approach with lighter-weight storage requirements is needed.

Recently, the ALACRITY [42, 48, 104] system presented an alternative indexing paradigm. Using a different data structure, the delta-compressed inverted index, ALACRITY demonstrates substantially lower index sizes vs. compressed bitmaps. Unfortunately, ALACRITY does not support query processing without decompression, and thus cannot take advantage of this memory savings and may incurring higher CPU costs. Also, the process to combine results from multiple ALACRITY indexes to answer multi-variate queries, though possible, is less natural than with bitmap indexes.

Ideally, then, we would like to develop an indexing methodology that maintains the beneficial decompression-less, multi-variate query processing properties of the compressed bitmap index, but with a significantly lower storage footprint.

In response, this chapter presents our work in developing a new form of index: the *Hyperdyadic Tree* (or HD-tree) index. Distinct from both bitmap and inverted indexing, the HD-tree index exhibits substantially smaller index sizes than an equivalent WAH-compressed bitmap index (by a factor of 1.14x to 1.90x in our experiments on real scientific data), making significant progress in reducing storage footprint. Nevertheless, the HD-tree index achieves comparable or superior performance versus WAH across most queries in our testing, and supports the same decompression-less and multi-variate query processing features as compressed bitmaps.

The HD-tree index is based on a novel use of the classic *quadtree* data structure. To briefly review, the quadtree is a tree structure whose nodes represent a recursive quadrant-wise subdivision of a spatial region; a more detailed recap is given in the next section. The quadtree has some desirable properties, including support for efficient set operation algorithms [70], and has many potential applications, such as monochrome image compression [51]. The HD-tree is a data structure based on the quadtree, with some differences as detailed in Section 2.3; in particular, the HD-tree uses a pointerless, "flat" encoding for straightforward storage on disk.

Our key approach is to develop an indexing approach using the same overall structure as a bitmap index, but instead of storing one bitmap per unique data value, we use one *HD-tree* per unique value. As we will demonstrate, the good image compression properties of quadtrees translate

into a storage-lightweight index structure. Furthermore, the key properties of the bitmap index are maintained: HD-trees can be manipulated without any form of decompression during query processing, and the evaluation of multi-variate queries over HD-tree indexes is straightforward.

In Section 2.3, we describe the HD-tree in detail, as well as several new related algorithms necessary to achieve the properties described above. First, however, we review some key background concepts that underpin our methodology.

2.2 Background

2.2.1 Compressed Bitmap Indexing

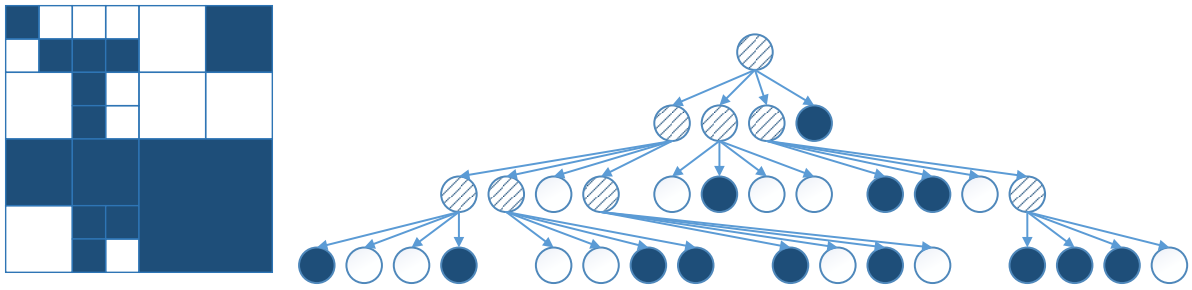
Traditional DBMSs have a long history of indexing methodologies, such the classic B-tree and its variants [24]. However, these methods are not well suited to multi-dimensional queries over high-dimensional, read-mostly scientific datasets [79]. Instead, bitmap indexes have been shown to be more effective for these types of queries and datasets [60, 92], thus bitmap-based databases such as FastBit [91] have become the standard for indexing scientific data. Bitmap indexing has successfully been applied on extreme-scale scientific data [14, 23].

2.2.2 Quadtrees

In its most abstract form, the "quadtree" is simply a tree in which every *internal* node has exactly four children (hence "quad"). The quadtree is most commonly used to represent a *recursive quadrant-wise decomposition of space*. Specifically, each node represents a certain region, with the root node representing the entire space, and the children of a given node representing the four quadrants of the space represented by their parent. This concept can be easily generalized to higher dimensions, with the "octree" being the 3D variant.

Interestingly, there is no widely-established term for a k-dimensional quadtree. Common candidates include the *hyperoctree* and the *orthtree* (from *orthant*, the k-dimensional generalization of a quadrant); however, neither is widely used in the literature. Instead, we adopt the convention of referring to the general case simply as a *quadtree*, making the dimensionality explicit as needed (e.g., 2D quadtree, k-D quadtree, etc.).

The particular type of quadtree of interest in this work is the *region quadtree*, used to describe (in the 2D case) a square, monochrome image with power-of-two side length (this too may be generalized to any dimensionality). In a region quadtree, the root node represents the entire image, which its descendant nodes representing a recursive quadrant-wise decomposition, with nodes at the lowest possible level of the tree representing individual pixels. Furthermore, each node has a



(a) An example 2D monochrome image

(b) The corresponding 2D region quadtree

Figure 2.1 An example of a (2D) region quadtree corresponding an example monochrome image. Filled nodes are "black," empty "white,", and striped "gray."

.

"color": one of black, white, or gray. Leaf nodes are always either black or white, indicating that the entire region they represent is either black or white, respectively. Every internal node bears the color gray, indicating a region of mixed color that will be further resolved the node's children.

An example of a 2D region quadtree is given in Figure 2.1, with Figure 2.1b showing the quadtree representing the 2D monochrome image in Figure 2.1a.

An detailed survey paper exists [70] explaining region quadtrees, among many other variants and application of the quadtree.

2.2.3 Linear Quadtrees

Quadtrees are a very useful in-memory data structure, but when it comes to serialization for storage, the common pointer-based implementation is unwieldly. Instead, various "linear quadtree" methods have been developed to encoding a quadtree in an encoded, pointerless form that is still amenable to various kinds of useful manipulations, such as set operations.

One of the first linear quadtree approaches, known alternately Gargantini coding [30] or tesseral addressing [7], operates by encoding each black node in a region quadtree as an integer code representing its path from the root of the quadtree; white and gray nodes are not stored, instead being understood implicitly. The *depth-first traversal expression* (or *DF-expression*) [36, 44] is another early alternative (though not originally developed as a quadtree encoding): a sequence of codes ("0", "1", or ")") are used to convey a depth-first traversal of a quadtree. Both of these methods are *depth first* in nature. The previously-mentioned quadtree survey paper [70] touches on these and other related depth-first approaches.

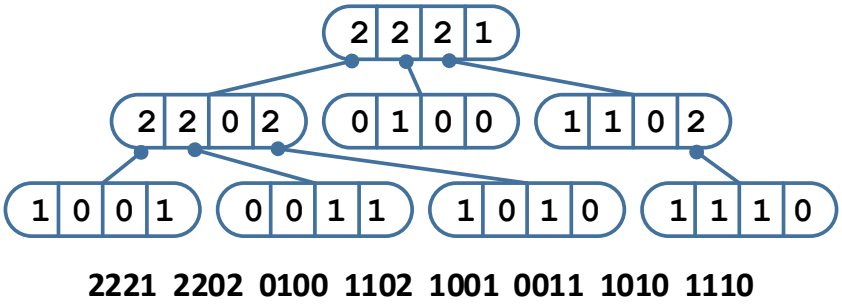


Figure 2.2 A CBLQ coding corresponding to Figure 2.1, depicted both in its conceptual form (top) and as stored in a series of code words in memory/on disk (shown in base 4) (bottom).

Somewhat later, various *breadth-first* traversal approaches were developed. Early approaches focused more on compression and progressive transmission [20, 38, 71]. Another survey covers some of these additional methods [37]. Later on, the Constant-Bit Linear Quadtree was presented [51], notable as a breadth-first approach with an emphasis on fast set operations.

Other related work includes improving CBLQ compression via Huffman coding [50], and various transformations between CBLQs and other breadth-first representations [19], depth-/breadth-first codings [21], and different quadtree dimensionalities [18].

The CBLQ is of special note, as it is an critical basis for our formulation of the HD-tree; therefore, we explain it in more depth next.

2.2.4 Constant-bit Linear Quadtree (CBLQ)

The CBLQ is a linear quadtree encoding developed for the compression of monochrome images [51]. It is a breadth-first encoding of a quadtree, and operates by representing each *internal* node as a *code word* consisting of four 2-bit *codes*; describing the colors of the *children* of that node (not the ndoe itself). Leaf nodes are represented implicitly, as children of their parents, and the root node is assumed to be gray. Each code may be either 0, 1, or 2, corresponding to a white, black, or gray child, respectively (technically, the original CBLQ also used a 3-code, but it is not useful in this work, and so we ignore it for simplicity). A CBLQ word therefore consists of one byte containing four 2-bit codes packed together. Figure 2.2 depicts a CBLQ coding of the quadtree and monochrome image from Figure 2.1.

The original paper also outlines an algorithm for set operations on pairs of CBLQs; this is critical in our development of the HD-tree for indexing and query. The algorithm walks through both CBLQs simultaneously, guided by an *action queue* indicating how to process the visited CBLQ words. Each

Table 2.1 The action table for the "union" action, as adapted from the original CBLQ paper [51]. "∅" indicates a no-op: no next action to be enqueued.

Left code	0			1			2		
Right code	0	1	2	0	1	2	0	1	2
Output code	0	1	2	1	1	1	2	1	2
Next action	∅	∅	copy-R	∅	∅	delete-R	copy-L	delete-L	union

action in the queue apply to the next word(s) in either the left, the right, or both CBLQs. The result of each action is determined by applying a corresponding *action table* to each code (for a left- or right-CBLQ action) or pair of codes (for a both-CBLQs action): each input code(s) is mapped to both an output code to emit and a next action to enqueue. The queue begins with a single action corresponding to the set operation to evaluate (e.g., "union"), and the algorithm terminates when the queue is empty.

For example, the "union" action applies to both CBLQs, and has the action table shown in Table 2.1. If the next left and right CBLQ words were 2012 and 1202, respectively, the output word would be 1212, and actions "delete-L," "copy-R," and "union" would be enqueued (the third pair of codes yields a no-op action, which is is not enqueued). For reference, the copy/delete-L/R action tables are set up to recursively copy/delete a given child's subtree in the left/right CBLQ, respectively. Working through this example, we see that the union of a 0-code and 2-code results in "copy" action, because the full content of the subtree rooted at the 2-code should appear in the output after union with an empty region (0-code). Likewise, the union of a 1-code and 2-code results in the deletion of the 2-code subtree, which is covered by a filled region (1-code). The union of two 2-codes requires further examination of the subtrees. Similar transition tables can be constructed for complement, intersection, set difference (i.e. AND NOT), and symmetric difference (i.e., XOR).

The key property of this algorithm is that it can combine two CBLQs and produce an output CBLQ *without decoding to intermediate, pointer-based quadtrees* at any point. This "decompression-less" property is key to our use of HD-trees as an alternative to WAH-compressed bitmaps.

2.2.5 Quadtrees in Value Indexing

Quadtrees and octrees have long been used as the basis for many spatial indexing techniques [70, 72]. However, we are unaware of any *value-oriented* indexing method (as opposed to spatial indexing) for array data based on quadtrees. The nearest analogue we have found is the use of k^2-trees to index and query raster data for GIS systems [8]; the approach used in that work (data structures, indexing and query algorithms, etc.) significantly differs from our own, however.

2.3 Method - The HD-tree Index

Our indexing approach is based on two key pieces: first, the HD-tree data structure itself, and second, how the HD-tree may be used to build an index. We explain these points in turn.

Recall that a bitmap index operates by storing a (possibly compressed) bitmap for each unique value in a dataset. For a bitmap associated with a given unique value, the positions of set bits indicate the record IDs matching that value. An equality query (e.g., $X = 10$) is answered by retrieving the bitmap for the equality value and converting to record IDs. Likewise, a range query (e.g., $10 < X < 20$) is evaluated by taking all bitmaps for values in the range and computing their bitwise OR, then converting this final bitmap to record IDs.

We propose to use the HD-trees, quadtree-like structures, in place of bitmaps in this scheme, one per bitmap. Our intuition for the use of quadtrees in place of bitmaps is two-fold. First, quadtrees are known to achieve good compression for many use cases (such as monochrome images), and so hold promise as the basis for storage-lightweight indexing. Second, efficient set operation algorithms exist for quadtrees [51, 70]. This is important during range queries, where the union of multiple quadtrees would need to be computed (i.e., the equivalent of a bitwise OR for bitmaps).

However, at first glance, two issues would seem to disqualify quadtrees for use in non-spatial, value-based indexing such as we consider here. First, quadtrees operate in a 2D domain, not the 1D domain of RIDs needed here, and so are not applicable as a replacement for 1D bitmaps. Second, the traditional pointer-based quadtree structure is not amenable to serialization on disk, a critical requirement for the type of index being developed here.

Instead, we propose the hyperdyadic tree (or HD-tree) data structure, which, while closely related to the quadtree, addresses these two concerns. The HD-tree is actually a class of data structures, parameterized with a positive integer k (i.e., a kHD-tree). A kHD-tree represents a tree that is *structurally identical* to k-dimensional quadtree: each node is either a leaf, or has exactly 2^k children. However, the *interpretation* and *storage* of an HD-tree differ from those of a quadtree, which is the key to addressing the aforementioned limitations.

The interpretation of an HD-tree is as follows. In contrast to a k-dimensional quadtree, which is understood to recursively subdivide a *k-dimensional space* into k-dimensional "hyperquadrants," a kHD-tree instead is seen to recursively subdivide a *one-dimensional space* into 2^k equal-length slices each step. These slices are each of length 2^{kj} for progressively decreasing non-negative integers j. For instance, a 2HD-tree subdivides a 64-element space into intervals (slices) of length $2^{(2\cdot 2)} = 16$, then $2^{(2\cdot 1)} = 4$, then finally $2^{(2\cdot 0)} = 1$ at the lowest level. Figure 2.3 depicts an example of this scenario. Intervals of this form are known as *dyadic intervals*, and the parameter k is analogous to the dimensionality of a k-dimensional quadtree; hence, we term this structure a *hyperdyadic tree*.

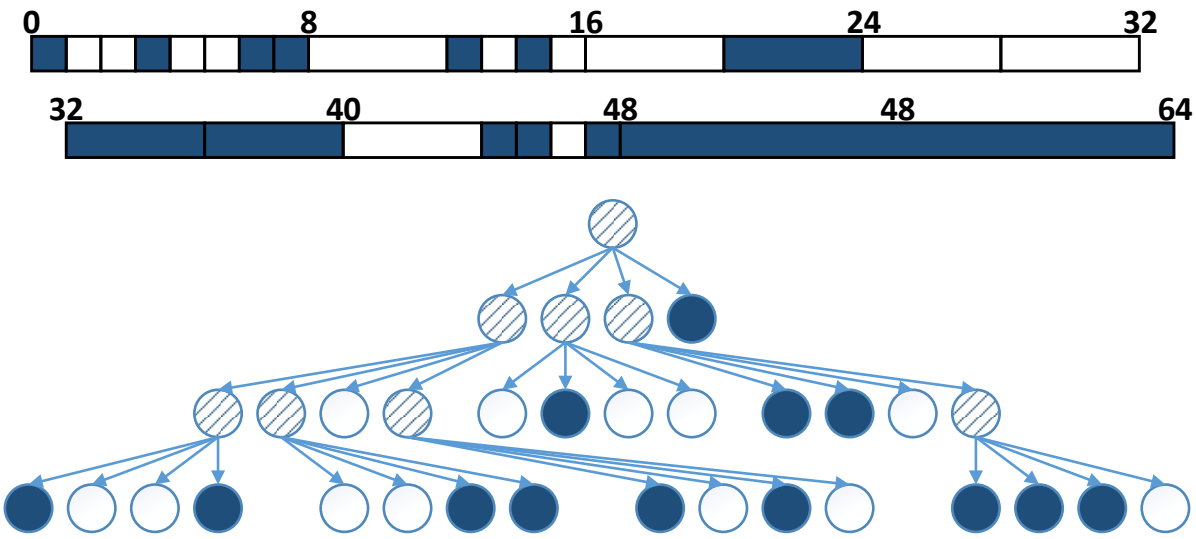


Figure 2.3 A 2HD-tree representing the contents of a 64-element 1D voxel space. The root node refers to the entire voxel space, whereas each child node describes a progressively smaller dyadic interval. "White" nodes indicate empty intervals, "black" nodes filled intervals, and "gray" nodes (striped here) indicate a mix of empty and filled voxels.

In terms of storage, the traditional quadtree is pointer-based, which is difficult to serialize. Therefore, we adapt the Constant-Bit Linear Quadtree (CBLQ) pointerless encoding as the basis for storing and manipulating HD-trees, generalizing it to k dimensions (rather than only 2D, as described in the original paper [51]). The CBLQ-encoded version of Figure 2.3 can be seen in Figure 2.4.

However, it turns out that CBLQ-coded HD-trees yield lackluster performance when using the algorithms and encoding described by the original paper. This is unsurprising, as the CBLQ was designed for image compression. In this section, we identify and address four areas in which we differentiate the algorithms and encoding of the HD-tree, optimizing it for indexing and query processing: HD-tree compression (Section 2.3.1), bulk construction of many HD-trees (Section 2.3.2), bulk set operations between many HD-trees (Section 2.3.3), and conversion from HD-tree to RIDs (Section 2.3.4).

2.3.1 HD-tree Dense Suffix Coding

The original CBLQ uses 2-bit codes in every word. However, for our HD-tree coding, we note that words in the lowest level never have internal nodes as children, and therefore consist of only 0-codes and 1-codes (the 2-code indicates the existence of a child internal node). This can be seen in

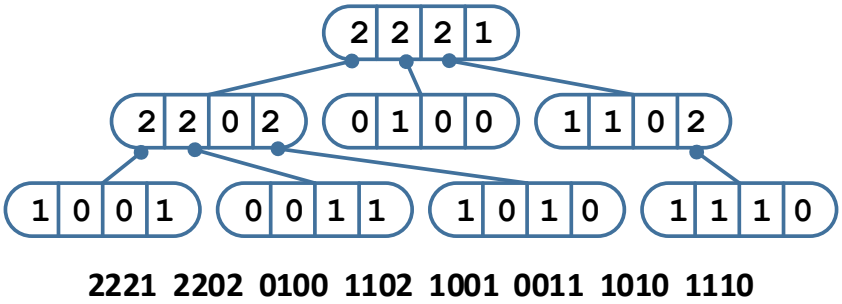


Figure 2.4 The HD-tree from Figure 2.3 encoded using the CBLQ coding method, depicted both in its conceptual form (top) and as stored in memory/on disk as integer words (shown in base 4) (bottom).

Figure 2.4, for example. Thus, our implementation eliminates this wasted space by using one-bit codes at the lowest level, which we call the *dense suffix* of the coded HD-tree.

Besides yielding a significant space reduction, dense suffix coding continues to support existing set operation algorithms with minimal modification. This will be described in greater detail in section 2.3.3. However, the key factor is that the set operation algorithms process HD-trees one level at a time, and this is the same level across all HD-trees involves in the operation; therefore the code for the final level can simply be replaced bit-wise operations vs. 2-bit code-based operations.

A similar observation may be made about some prefix of HD-tree words: typically, only 0- and 2-codes will appear until lower down in the tree, opening an opportunity for a *dense prefix*. However, such dense prefixes are irregular, not aligned to levels like dense suffixes, and so interfere with set operations. While dense prefixes may be useful for reduced storage at the cost of decompression time, at this time we only implement dense suffixes.

2.3.2 Bulk HD-tree Construction

In the seminal paper, a CBLQ is generated from a pre-existing n-bit bitmap in $\Theta(n)$ time. This is acceptable in the context of (monochrome) image compression, where only one CBLQ need be built. However, when translated to index building, one HD-tree must be built *per bin*, leading to a complexity of $\Theta(nb)$, where the number of bins b may be in the thousands or more. This construction method is thus intractable in this context.

A key insight into an alternative is that the per-bin HD-trees contain relatively few RIDs (i.e., consist predominantly empty/white space). Indeed, for a dataset of n points and b bins, exactly n RIDs are scattered among nb total possible RIDs that could be represented in these b HD-trees. A more efficient HD-tree construction algorithm, then, could skip over these empty runs in bulk,

rather than examining every individual RID for every HD-tree (as the original algorithm implicitly does).

Therefore, we consider a method for pushing *runs* of absent/present RIDs into an incrementally-built HD-tree. This allows the index to be built with a single scan of the input data. During this scan, runs of RIDs falling into the same bin are collected, with each run being pushed all at once as a single unit into the bin's corresponding HD-tree. With this approach, at most n such runs will be pushed across all RSets.

We implement this run-pushing ability using a structure we call the *HD-tree builder*. When building a kHD-tree over a dataset of size n, which will result in an HD-tree of $\ell = \lceil \frac{1}{k} \log_2 n \rceil$ levels, this structure consists of two parts:

- `levels`: an array of length ℓ of lists of HD-tree words[1], each initially empty; and
- `pwords`: an array of of length ℓ of partial HD-tree words (i.e., small lists holding between 0 and $2^k - 1$ HD-tree codes), each an initially-empty partial word

The HD-tree builder exposes a single function, `push_run(count, is_filled)`, a call to which indicates the next `count` RIDs should be recorded as either either present or absent in the HD-tree (depending on the boolean value of `is_filled`). An "absent" RID push is used to "catch up" an HD-tree builder to the current RID position before pushing a run of "present" RIDs. The `push_run` implementation is given in Algorithm 1. After a series of calls with total `count` of n, the final HD-tree may be retrieved by concatenating lists `levels[0]` through `levels[`$\ell-1$`]`.

Though a single invocation of `push_run` is $O(\ell) = O(\log n)$, the worst-case amortized time for c calls with total `count` of n can be shown to be $O(c \log \frac{n}{c})$. The intuition behind this bound is as follows: while any single push may be $O(\log n)$ due to all partial words being "almost full," this worst-case state cannot occur repeatedly over many pushes, and so the complexity per push of length p tends to $O(\log p)$. The total cost of pushes $p_1, \ldots, p_c$ is then $\sum_i \log p_i = \log \prod_i p_i$. Since $\sum_i p_i = n$, this cost is maximized when $p_1, \ldots, p_c = n/c$.

It can be further shown that, under this per-bin worst case complexity, the overall worst case time for b bins is to spread pushes round-robin across all bins. This yields a per-bin complexity of $O(\frac{n}{b} \log \frac{n}{n/b})) = O(\frac{n}{b} \log b)$, and thus a total amortized complexity of $O(n \log b)$, far superior to the $\Theta(n\,b)$ time yielded by straightforward application of the original algorithm.

2.3.3 Bulk HD-tree Set Operations

While bulk construction makes building an HD-tree index feasible, query evaluation runs into its own bottleneck: set operations, especially union. Set operations on HD-trees are key to producing a

[1] We use the terms "array" and "list" here to refer to "fixed-sized array" and "dynamically-sized array", respectively

Algorithm 1 HD-tree builder pseudocode

```
State: k ← parameter for k HD-tree
State: ℓ ← number of HD-tree levels expected
State: levels ← new array of lists[ℓ]
State: pwords ← new array of partial words
  function PUSH_RUN(integer count, bool is_filled)
    code ← 1 if is_filled, else 0
    for i ← ℓ−1...0 do
      append ← min(count, 2^k − pwords[i].ncodes())
      Append append (code)-codes to pwords[i]
      count ← count − append
      if pwords[i].ncodes() = 2^k and i > 1 then
        if pwords[i] is all 0-codes or all 1-codes then
          Append one (code)-code to pwords[i−1]
        else
          Append one 2-code to pwords[i−1]
          Append pwords[i] to levels[i]
        end if
        pwords[i].clear()
        extra ← count mod 2^k
        Append extra of (code)-codes to pwords[i]
        count ← count − extra
        ▷ note: continues to next iteration
      else
        return
      end if
    end for
  end function
```

single result from many per-bin HD-trees during query processing. Therefore, it is critical to ensure they are as efficient as possible.

The original CBLQ paper presents a generic algorithm to evaluate any binary set operation on a pair of CBLQs. Yet, when adapted and used to combine more than a few HD-trees during query processing, performance suffers relative to the bitwise OR of equivalent WAH-compressed bitmaps. In particular, the "union" operation becomes a bottleneck, as most queries require merging many bins. Thus, a more efficient HD-tree set operations algorithm is needed.

As detailed earlier in Section 2.2, the original CBLQ set operations algorithm processes a pair of CBLQs at a time, using an *action queue*-based breadth-first traversal [51]. This algorithm walks through both CBLQs in tandem, producing an output CBLQ along the way.

Though this algorithm can be adapted to HD-trees for n-ary set operations via a binary expression tree, it is often inefficient. First, it is not cache friendly with many operands: an expression tree involves jumping between many pairs of HD-trees, reducing the benefit of caching from previous operations. Second, for n operands, a total of $2n-2$ HD-trees are traversed, and the intermediate HD-trees get progressively larger and more complex higher up the binary expression tree as they are combined. Ideally, only the original n HD-trees would be traversed, with no expensive intermediate HD-tree(s) to process.

We propose a radically different approach, which exhibits better cache efficiency and involves neither redundant processing of HD-trees nor the creation of any intermediate results. As opposed to the *binary, word-by-word traversal* employed by the original algorithm to process HD-trees in pairs, our *n-ary, level-by-level traversal* algorithm processes n operand HD-trees simultaneously (eliminating redundant processing and intermediate HD-trees), and interleaves the processing of operands on a tree level-by-level basis (for greater cache friendliness).

The algorithm's details are difficult to explain via pseudocode due to the amount of bookkeeping involved. Instead, we provide Figure 2.5 to depict simple run of the algorithm to union three 2HD-trees of height 2, covering a voxel space of 16 elements (also shown). The algorithm is inductive in nature: for each level, the number of output words, initial actions to apply to those words ("actions"), and the mapping from operand input words to output words ("mappings[]") are all assumed to be known. Based on this information, all words from the current level across all input operands ("operands[]") are consumed and applied to the current level of the output HD-tree ("output"), and the antecedent information for the next level is computed. The initial state (i.e., the inductive base case) is trivial to compute: exactly one output word, the initial set operation in question applies to it, and the single input word from each operand maps to the single output word.

As a final refinement, we have also developed a hard-coded n-ary *union* algorithm, since union is so heavily used in, e.g., equality encoding. It follows the same principles as the general n-ary algo-

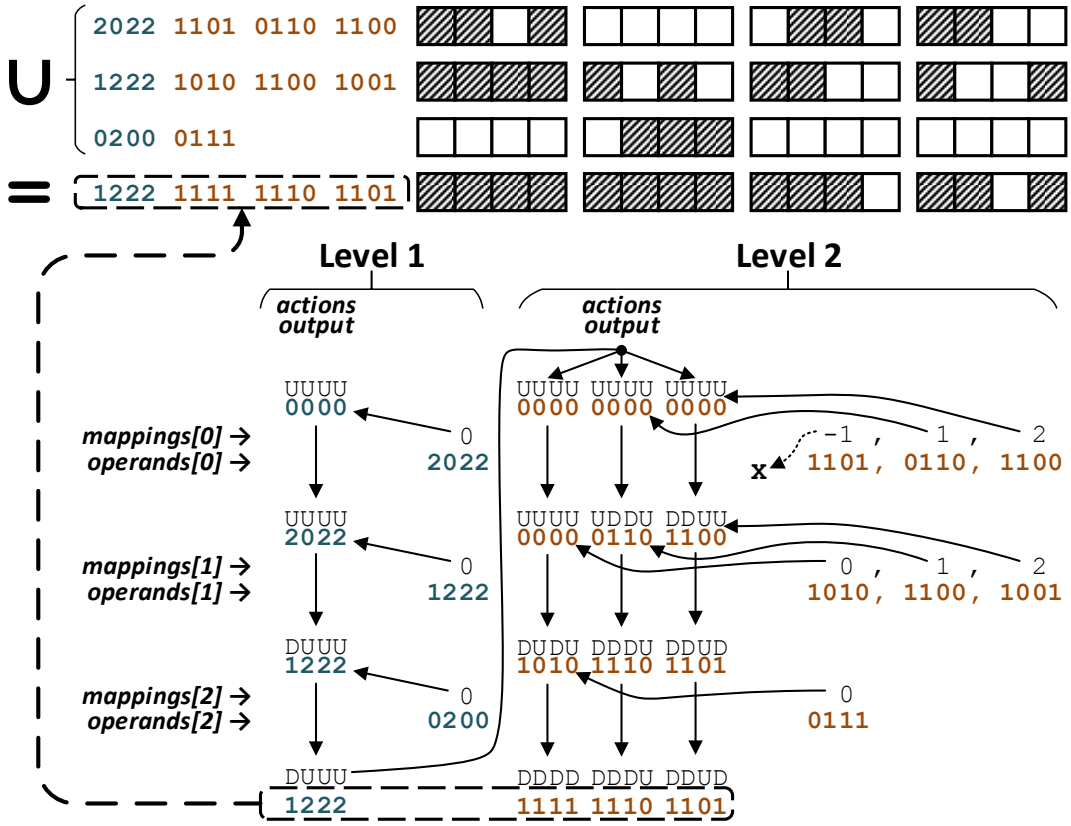


Figure 2.5 A depiction of our n-ary HD-tree set operation algorithm, as applied to compute the union of three 2HD-trees.

rithm, but forgoes a transition table for bit-wise operations, and as well as a few other optimizations. This final algorithm generally outperforms equivalent an WAH bitwise-OR operation, especially as the number of operands (e.g., bins) increases.

2.3.4 Efficient HD-tree-to-RID Conversion

The final key algorithm for handling HD-trees in indexing and query is conversion to RID list. While query processing uses only HD-trees internally, the user will generally want query results in RID list form, and other functions, such as candidate checks, may also rely on RID lists. To our knowledge, however, no algorithm for RID conversion (or equivalent problem) for CBLQs has proposed in the literature.

Thus, in developing a suitable conversion algorithm for HD-trees, we first apply the natural

approach of a *breadth-first traversal*, since HD-trees are breadth-first structures. The algorithm is similar to that for set operations (though it is a unary operation, rather than binary): maintain a queue of current RID offsets, initialized to a single element, 0. Each iteration, pop an RID offset and the next kHD-tree: the offset corresponds to the offset of that word's dyadic interval. Each 1-code encountered emits a run of RIDs (determined by the popped offset, code position, and current tree level), whereas each 2-code enqueues a new offset (determined by the same); 0-codes are ignored.

While this algorithm works, we have developed a second approach. Counter-intuitively, a *depth-first* traversal of the breadth-first HD-tree structure ends up achieving faster conversion. In order to iterate the "wrong way," the algorithm uses the starting offset of each tree level in the HD-tree word array, computed at HD-tree construction time, to initialize a list of per-level pointers, which it then "walks" left-to-right across the tree. This depth-first algorithm is given in Algorithm 2.

Algorithm 2 Depth-first HD-tree to RID conversion

function RIDCONVDFHELPER($k, \ell, level, b, ptrs, out$)
  $s \leftarrow 2^{k(\ell - level)}$
  $word \leftarrow$ `dereference`$\big(ptrs[level]\big)$
  Increment $ptrs[level]$
  for i in $0 \ldots 2^k - 1$ **do**
    $c, p \leftarrow word[i], b + i \cdot s$
    if $c = 1$ **then**
      Append $p, \ldots, p + s - 1$ to out
    else if $code = 2$ **then**
      `RIDConvDFHelper`$\big(k, \ell, level + 1, p, ptrs, out\big)$
    end if
  end for
  return out
end function

In: $octree$: a kHD-tree with ℓ levels
function RIDCONVDF($octree$)
  $ptrs \leftarrow$ pointers to level starts in $octree$
  return `RIDConvHelper`$(k, \ell, 1, 0, ptrs,$`new RID list`$)$
end function

At first glance, this recursive algorithm does not appear efficient, as the call to `RIDConvDFHelper` is not eligible for tail recursion elimination. However, the maximum recursion depth is limited by the max HD-tree height, which itself is tightly bounded: $\lfloor 64/k \rfloor$, assuming 64-bit RIDs (any higher

and the HD-tree could overflow the RID type). Given that the main loop body is quite short, it is possible (for the compiler) to *fully flatten* this recursion, yielding a deeply-nested loop (in machine code). The function may then be instantiated for desired values of k, after which the compiler can reasonably convert most state to registers (b, $ptrs$) or hard-coded constants (k). Due to these possible optimizations, we observe this depth-first version to be faster than the breadth-first variant in practice.

2.4 Results

To evaluate the HD-tree, we compare its performance against the widely-used FastBit implementation of WAH-compressed bitmaps (technically FastQuery [23]; see next section). First, we take large-scale real-world scientific data ($\sim$ 3 TB) and index multiple data variables, measuring the index size generated by both approaches. Then, we perform a range of queries over both index implementations to measure query response time.

We also explore the impact of *binning with different granularities* on both index size and query time. Binning effectively changes the data cardinality while roughly maintaining the bin distribution, giving us a sense of the impact of cardinality on index size when using compressed indexing approaches.

2.4.1 Experimental Setup

All of our experiments were run on the Edison supercomputer at the National Energy Research and Scientific Computing Center (NERSC) [56]. Each node contains 24 cores with 64 GB shared memory. For our serial experiments, we always reserve a full 24-core Edison node in order to eliminate interference from co-resident workloads. Edison's Lustre parallel filesystems were used for storage in each experiment, specifically the "/scratch1" and "/scratch2" filesystems, each consisting of 96 Lustre OSTs providing a total of 2.1 PB of storage and 48 GB/s aggregate peak I/O performance [26, 57] (though we used two filesystems due to storage quota limitations, we ensured each experiment was performed entirely on one or the other).

We run our evaluations using a large-scale dataset from VPIC [9–11], a three-dimensional electromagnetic relativistic kinetic plasma simulation code, consisting of 7 single-precision floating-point variables: energy, x/y/z (position), and Ux/Uy/Uz (velocity components relative to the underlying magnetic field). However, we only use the first four in our experiments. Additionally, based on precedent [14], we use a pre-filtered subset consisting of "interesting" particles, defined by $energy \geq 1.1$, yielding 180 billion particles and a 5 TB dataset ($\tilde{7}$50 GB per variable).

Since the data are floating point, it is necessary to employ binning. We use decimal-precision binning as developed and used in FastBit [87], in order to ensure comparability with our system. Under precision-d binning, values are binned together when equal after truncating at d digits of (decimal) precision (i.e., d significant digits). For example, the values 1.31×10^3 and 1.34×10^3 would be binned together under precision-2, but not under precision-3 or higher.

For software, we use the FastQuery [23] indexing and query system, which is a layer on top of FastBit [88, 91] (developed by the same organization) that provides additional features, which will be useful for comparison in later chapters. Since they are so closely related, we use the terms FastBit and FastQuery interchangeably. FastQuery version 0.8.2.8 and FastBit version 1.3.9 were used, the latest versions available at the time we began our experiments. However, we do fix one important bug in FastBit 1.3.9: in our early experiments, FastBit was loading the entire index into memory for each variable queried, even when only a small fraction was needed. We have reported and confirmed this issue with the developers; correcting this bug significantly improved I/O times for FastBit (and FastQuery) in our tests.

2.4.1.1 A Note on Comparison with WAH

It should be noted that two ways exist to conduct the comparison between HD-tree and WAH indexing. First, it is possible to compare our HD-tree implementation to the FastBit indexing software. However, this might cloud the comparison with confounding implementation factors, such unequal code optimization, I/O library used, etc. Alternatively, we could use our codebase[2] as the platform for both HD-tree and WAH indexing and query, borrowing only the core WAH bitmap compression code from FastBit. This equalizes most implementation details, but could conceivably disadvantage WAH if any optimizations from FastBit were not correctly ported.

Since both approaches have advantages, we employ *both* side-by-side in our analysis. For index size, we have verified that no different exists between these approaches, and only one result for WAH is reported. For query performance, however, we report both: "WAH" indicates WAH within our framework, whereas "FastQuery" timings are collected from running queries in standalone FastQuery.

2.4.2 Indexing Performance

Our first experiment compares the indexing performance yielded by 3HD-tree, 4HD-tree, and WAH on four VPIC variables. We repeat the experiment once each using "precision-3" and "precision-4"

[2]This option is made possible by our unified indexing and query engine, PIQUE, which we develop in Chapter 3. PIQUE will be discussed in detail in that later chapter; here, however, the details of this system are not important yet.

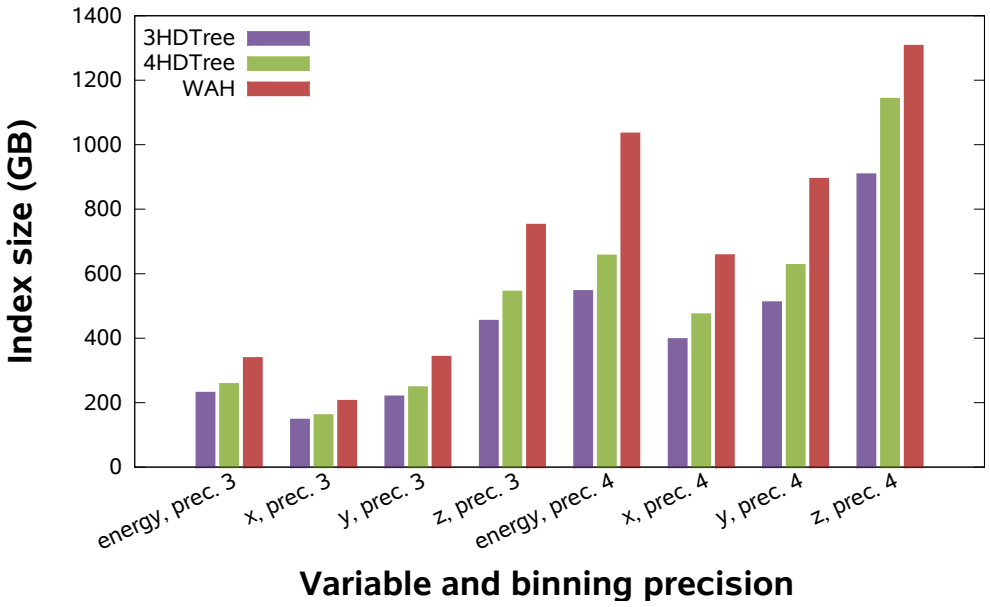


Figure 2.6 Index storage footprint vs. variable/binning method for WAH, 3HD-tree, and 4HD-tree. Two trends are notable: 1) for any given variable/binning method, index storage size follows 3HD-tree < 4HD-tree < WAH; and 2) different variables and binning precision are more or less space-costly to index than others, despite the input data being the same size in bytes in each case.

binning, for a total of eight variable/binning cases.

We measure the resultant index size for each method, as reported in Figure 2.6. The most notable trend in these results is that, for storage size vs. index method, 3HD-tree < 4HD-tree < WAH holds in these experiments. Specifically, the WAH index is between 1.30x and 1.90x larger than the 3HD-tree index and between 1.14x and 1.58x larger than 4HD-tree across these cases. This implies that the HD-tree representations are significantly more compact than WAH in practice; we reinforce these results with a theoretical model for expected index size later in Chapter 4.

It is also notable that index storage space varies with the variable and binning method used, regardless of whether WAH bitmaps or HD-trees are used. This trend correlates very strongly with the *number of bins* produced, as reported in Table 2.2. For all variables, precision-4 binning yields about 7-9x as many bins as precision-3, explaining the higher storage cost in these cases.

Surprisingly, variable `z` yields over 10x more bins than any other variable. The cause, it turns out, is central value (e.g., mean): `x` and `y` are centered around roughly $+160$ and -160, respectively, whereas `z` is centered at 0. The IEEE floating-point format concentrates exponentially more values

Table 2.2 Average bins per index partition for different variables/decimal-precision binning methods. Note the strong correlation with index storage footprint in Figure 2.6.

Variable	Bins/Part. (prec. 3)	Bins/Part. (prec. 4)
energy	155.4	1123.2
x	98.8	816.5
y	115.0	957.8
z	4260.4	33718.0

at small absolute values (e.g., roughly the same number of values lie in $(-1,1)$ as in $(-\infty,1]\cup[1,\infty)$)[3]. Thus, very small deviations around 0 give rise to many new bins, and so this effect is endemic to "exponential" binning methods. Further analysis is outside the scope of this work, yet it remains an important phenomenon that should be considered when indexing scientific data.

To conclude, we infer that the HD-tree is a more compact indexing method than WAH, by 1.14x to 1.90x across these experiments. Additionally, we have demonstrated how the nature of data to be indexed, combined with binning choice, has a large impact on index size regardless of the type of index used.

2.4.3 Query Performance

We now turn to query performance. In order to facilitate more comprehensive experiments, indexes for this test are built over a 2^{35} element subset of the VPIC "energy" variable (= 128 GB, or about 1/6 of the original variable). All queries are index-only queries, returning a list of RIDs answering the query constraint. Reported query times include 1) index reading to retrieve necessary HD-trees or WAH bitmaps, 2) the "index decoding" step (i.e., taking the bitwise-OR/union of loaded bitmaps/HD-trees), and 3) converting the result bitmap/HD-tree to an RID list. It does not, however, include time to write these result RIDs to disk, since this task is identical across indexing methods and would introduce unnecessary timing variability (also, this last step is not always necessary, as some applications might use in-memory processing on the results).

In comparing WAH and HD-tree indexes, we consider two index parameters as experimental factors. First, since relative performance clearly depends on query selectivity, we vary query selectivity, from $\approx 33\%$ to $\approx 1.7\%$. Second, we expect binning to have significant impact; therefore, we repeat the experiment for both precision-3 and precision-4 binning. Figure 2.7 reports the experiment results, and includes total time to perform equivalent queries using FastQuery.

In terms of I/O, the comparison of bytes read between 3HD-tree/4HD-tree and WAH appears

[3]This fact is unrelated to cardinalities of sets of reals; the set of IEEE floating-point numbers is both discrete and finite.

complicated, but ultimately appears to be based on the *density* (i.e., fraction of RIDs present) of the bins used to answer the query. The HD-tree exhibits superior compression at most densities, but at very low densities WAH is marginally better. Bins in the precision-3 index are relatively more dense (due to coarser binning). Also, in this particular dataset, lower `energy` values are more common, leading to denser bins. Hence, the HD-tree methods induce less I/O for the lower-cardinality binning and for queries touching more densely-clustered portions of the value domain; in other cases, the amount of I/O is quite similar between HD-trees and WAH bitmaps.

A related trend is that 3HD-tree generally reads fewer bytes, but costs more CPU time, than 4HD-tree. This is reasonable, as 4HD-trees use larger words, which improve algorithmic efficiency but increase "internal fragmentation" by requiring longer runs of present/absent RIDs for compression.

The results also indicate both HD-tree methods generally require less CPU time than WAH. However, this trend reverses for low-selectivity queries on the precision-3 binned index, where WAH is faster. This behavior can be explained as follows. Individual binary set operations on WAH compressed bitmaps are faster than those on HD-trees, but WAH has no analogue of the n-ary HD-tree union algorithm, and thus pays a progressively higher relative penalty as more operands (i.e., bins) are introduced. Thus, a higher-precision index with more bins benefits HD-tree. Note that, counter-intuitively, higher-selectivity queries for this dataset (i.e. toward the right of the plot) actually induce *more* touched bins, due to complement-wise evaluation (see below); hence, HD-trees also do relatively well in both of these cases with more numerous bins.

There is one further detail on the above experiment. Any range query may be answered either *by direct evaluation* (returning touched bins) or *by complement* (returning the complement of *un*touched bins). In real-world use, a heuristic must select the predicted faster mode; here, however, an imperfect heuristic might invalidate our results. Therefore, we run every experiment in both direct mode and complement mode, then report timings from the mode yielding lower *total time* in each experiment. Complement mode is active in many $energy \geq 1.15$ cases and a few $energy \geq 1.2$ cases, with WAH benefiting from it most often, and 3HD-tree least. The transition to/from complement mode causes the inflection points seen in CPU time and I/O plots.

FastQuery appears noticeably slower than PIQUE (the Y-axis cuts off to preserve a detailed scale). Since PIQUE's code for WAH is borrowed directly from FastQuery, we believe the I/O model must differentiate the systems' performance (collective HDF5 I/O vs. independent I/O, respectively). Also, it may be that FastQuery performs expensive candidate checks, despite our attempts to prevent this by using bin-aligned queries.

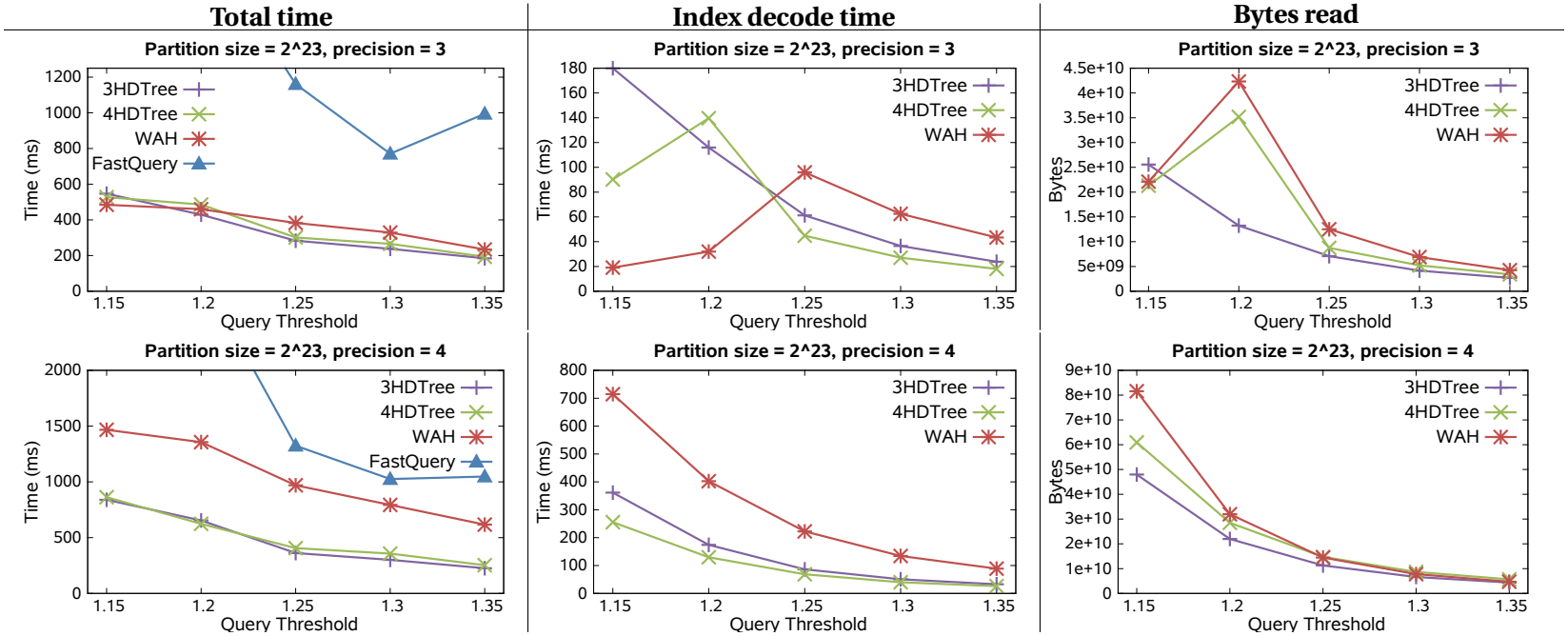


Figure 2.7 Query performance vs. query value threshold. Each plot reports a query statistic for $energy \geq T$ while varying $T \in \{1.15, 1.2, 1.25, 1.3, 1.35\}$ along the X axis. These thresholds (from left to right) roughly correspond to query selectivities 33%, 12%, 5.3%, 2.8%, 1.7% (for precision-4; precision-3 has slightly higher selectivity percentages due to how rounding is evaluated). Each query is evaluated two ways, *directly* and *by complement*, with the choice yielding minimum total time in each case being presented (see text). Plot columns represent the following query statistics (from left to right): total time, index decode time (i.e. CPU time), and bytes read from disk. Plot rows vary the binning precision between 3 and 4.

2.5 Conclusion

Motivated by the need for a more storage-lightweight alternative to the compressed bitmap index that still maintains its unique decompression-less query processing properties, we develop the HD-tree index. Based on a specialized linear quadtree encoding, we optimize the HD-tree for indexing and query processing with an improved compression encoding, a bulk construction algorithm, a bulk set operation algorithm, and an RID conversion algorithm. With these innovations, the HD-tree index demonstrates a substantial reduction in storage relative to the WAH-compressed bitmap index, while still maintaining or improving query performance in most cases.

As a final note, the observant (or perhaps skeptical) reader may question whether the HD-tree is truly a fundamentally different concept from the bitmap, or if it is instead simply a new form of bitmap compression. This observation is not without merit. In fact, this apparent parity contributes to the motivation of the next chapter, wherein we show (among other things) that bitmaps and HD-trees, along with ALACRITY inverted lists, may all be viewed as instances of a single, more fundamental concept: the *RSet representation*.

Chapter 3

Generalized Indexing and PIQUE

3.1 Introduction

The bitmap index [61] is the current go-to method for supporting query-driven exploration, visualization, and analytics for extreme-scale scientific data. Considered a distinct class of index unto itself (as opposed, for example, to the B+ tree and its variants), the bitmap index is known for its key ability to evaluate queries over many variables efficiently by combining results from per-variable indexes [80], and its good performance and (with compression) low storage footprint in many circumstances [92]. Many extensions and refinements have been explored in the literature, including binning [82], bitmap compression [4, 43, 90], and bitmap encodings [16, 17, 76, 94, 98].

An alternative index for scientific data was presented with the advent of the ALACRITY [42, 48, 104] index. ALACRITY employs a completely different data structure, the delta-compressed inverted index, and achieves substantially lower index sizes vs. compressed bitmaps. Yet, despite this key difference, the ALACRITY and bitmap indexes bear marked similarities, as well. To name a few: both approaches associate a data structure (inverted list or bitmap) with each unique value in a dataset, respond to queries via unions of these data structures, and may use binning to handle high data cardinality.

Now, in Chapter 2, we have presented the third alternative: new hyperdyadic tree index. Employing yet another distinct data structure, the HD-tree, it too bears useful performance characteristics: reduced storage versus compressed bitmap indexes while maintaining or improving query performance in many cases. Nonetheless, the HD-tree index is also similar in structure to the compressed bitmap index, in the same ways as ALACRITY is (HD-tree per unique value, union of HD-trees to answer queries, binning to reduce effective data cardinality).

These observed similarities in index structure raise the question, “Is there a generalized index

model that encompasses the bitmap, ALACRITY, and HD-tree indexes?" In this section, we present the first such model for what we term the *value-sliced index*, which also covers query processing. This model defines a broad design space in which these three indexes are instances, and is also general enough to cover all prior work in bitmap index compression, encodings, and binning known to the authors.

Besides contributing to the potential future development of value-sliced indexing methods, our generalized model also leads us to another, more concrete contribution: the development of the Parallel Indexing and Query Unified Engine (PIQUE). By realizing the abstract concepts of the value-sliced indexing model in a system implementation, PIQUE synthesizes disparate indexing techniques into one framework: from WAH bitmaps, indexing encodings, and precision binning ported from FastBit [91]; to ALACRITY inverted indexing and significant-bit binning; to our new HD-tree method. Furthermore, since our model decomposes value-sliced indexing into orthogonal stages, PIQUE is able to combine techniques from different sources in ways never before seen: for instance, PIQUE can build an index combining ALACRITY significant-bits binning, the HD-tree data structure, and (traditionally bitmap-only) interval encoding.

We conduct a performance evaluation of PIQUE, demonstrating parallel indexing scalability. In fact, we already conducted a performance evaluation of PIQUE in Chapter 2.4, as both indexing and query experiments were run on PIQUE. These tests also demonstrate the flexibility of the PIQUE system, a result of its solid theoretical foundation.

3.2 Background

3.2.1 Bitmap Indexing Techniques

Relational DBMSs have a long history of indexing methodologies, with the B+ tree and its variants being perhaps the most well-known [24]. However, for scientific data, where multi-variable queries over read-only/append-only data are commonplace, the bitmap index [60] is more effective [79, 92]. thus bitmap-based databases such as FastBit [91] have become the standard for indexing scientific data. Bitmap indexing has successfully been applied on extreme-scale scientific data [14, 23].

Many of the key developments in bitmap indexing have been in *compression*, *index encoding*, and *binning*. Bitmap compression techniques not only reduce index storage, but can also speed up query performance [93], and so are an integral part of modern bitmap indexing. Complementing compression to reduce bitmap size, index encodings compose and layer bitmaps to optimize for reduced space, query time, and/or query I/O, as well as for particular query types. Finally, binning methods limit the effective cardinality of the data (i.e., number of unique values), reducing index

size and making indexes on high-cardinality attributes tractable [82]. We discuss each of these techniques in turn.

Bitmap compression has evolved through several stages. Early compression of bitmap indexes relied on general-purposes compressors. Later methods instead focused on optimizing query performance; these include bitmap-optimized compression such as Byte-aligned Bitmap Compression [4, 5] and WAH [89, 90]. Many other later variants of BBC and WAH also exist; a recent comprehensive survey of this field exists [22].

Bitmap encoding is another key bitmap indexing technique. The basic bitmap index, where one bitmap is stored per unique value, is considered to be *equality encoded*. More complex encodings involve storing a set of bitmaps that does not directly represent the set of unique data values, but which can be combined via bitwise operations in various ways to answer any query posed. Range encoding [60] is perhaps the most straightforward: each bitmap represent all records *less than or equal to* a given value, rather than just equal to; range queries are evaluated by taking the bitwise difference (e.g., AND NOT) of the bitmaps corresponding to the endpoints of the query. The range encoding is further refined in the more compact interval encoding [17]. Both range and interval encoding are more storage-heavy than equality encoding (when compression is used), but significantly reduce query compute and I/O for range queries. Binary encoding [60, 85, 86, 98], later reclassified as base-2 component encoding [16], is an early example that offers significant overall index storage reduction at the cost of high per-query I/O cost. The more general base/component encoding [16] is a parameterized encoding that covers equality, range, binary, and other encodings as special cases. Multi-level encodings [76] combine multiple layers of existing encoding methods for better performance across multiple query types, at the incremental cost of storage. Extensive survey and performance studies exists comparing these and other encoding methodologies [94, 95].

Finally, we consider binning techniques. The idea of binning is to have one bitmap per *group of values* (i.e., bin), rather than per unique value. This can substantially reduce the number of bitmaps required, and therefore the overall size of the index [82, 92]. Many binning methods exist, including: (decimal-)precision bins [87], significant bit-based bins [42, 48], and query-driven overlapping bins [34]. The main drawback of binning is that the query response from a binned bitmap index is not always precise, as a query boundary may fall inside a bin (a so-called *boundary bin*), which may contain both matching and non-matching records. Resolving these *candidate records* requires the use of *candidate checking*, i.e., probing of the original data; the performance implications of this process, as well as ways to tune binning to minimize its cost, have been studied extensively [34, 65–67, 77, 81, 82, 96].

Many of these bitmap indexing techniques are realized in the FastBit indexing system [87, 88], which implementation is the basis for many of the studies cited above. The FastQuery [14, 23] system

is built on top of FastBit, adding parallel index building and querying, as well as additional services such as expanded interoperability with I/O middleware [33].

3.2.2 ALACRITY inverted indexing

Recently, inverted indexing techniques have been applied to scientific data, yielding substantially higher compression than compressed bitmaps [42]. Adapting techniques originally designed for web search and information retrieval [105], the ALACRITY compressed inverted index can fulfill many of the same use cases as bitmap indexing. Additional studies have explored in situ aggregation [48], as well as multi-variate query evaluation via a bitmap conversion/intersection technique [104].

3.2.3 Other Generalization Efforts in Indexing

Our contribution of generalizing the bitmap, ALACRITY, and now hyperdyadic tree indexing methods is unprecedented to our knowledge. However, such generalization efforts have precedent in the database community. For example, the GiST model generalizes much prior work in search tree-based indexes [35]. In another instance, the multi-component bitmap encoding [16] unifies equality, range, binary-component, and bit-sliced indexes, while also uncovering new bitmap index design choices.

In similar form, we generalize the bitmap index to encompass other index types previously considered unrelated. We then leverage this model to develop a unified indexing and query framework, PIQUE, and later on in Section 6.2, we explore some potential directions for novel indexing methods that our model highlights.

3.3 Formal Indexing Model

In order to define an indexing and querying model, we must first define our data. Note: hereafter, we use $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$, $\mathbb{N}_N = \{1, 2, \ldots, N\}$, $\mathcal{P}(X)$ the set of all subsets of X (i.e., the power set), and $I_S = \{[a,b], [a,b), (a,b], (a,b) \subseteq S\}$ the set of all intervals of a totally-ordered set S.

Definition 3.3.1. A *data variable* D of size n is a linearized list of n values from a given value domain set V with total order $\leq$ (e.g., $V = \mathbb{R}$). That is, D is a function $D : \mathbb{Z}_n \to V$, where $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. A *record ID* (or *RID*) $r \in \mathbb{Z}_n$ is a position within the data variable, and corresponds to a value $D(r) \in V$.

Given such a data variable, a user may ask which record IDs have values within a given range, for example, $[10, 20]$. Such a *range query* is answered by the *query function*:

Definition 3.3.2. The *query function* is a function $\mathcal{Q} : I_V \to \mathcal{P}(\mathbb{Z}_n)$ where $I_V \subset \mathcal{P}(V)$ is the set of all intervals in V, that is, $I_V = \{[a,b], [a,b), (a,b], (a,b) \subseteq V\}$. The query function maps a given *range*

query interval $q \subseteq I_V$ to the set of record IDs with values in that range, that is, $\mathcal{Q}(q) = \{r \in \mathbb{Z}_n \mid D(r) \in q\}$.

Note that *equality queries* are special cases of range queries, $[a,a] = \{a\} \subset V$, as are one-sided range queries, $[a, \text{inf}) \subset V$.

3.3.1 The Value-sliced Index

An index $\mathcal{I}$, then, is a structure that aids in computing the query function $\mathcal{Q}$. We define a *value-sliced index* $\mathcal{I}_{Q,R,E}$ over data variable D as the result of three orthogonal steps: a *quantization* (or *binning method*) Q, an *RSet representation* R, and an *index encoding* (or just *encoding*) E. Informally, a quantization maps data values to a set of *quantized keys*, partitioning the data variable into *bins* (sets of record IDs sharing the same quantized key)[1]. An RSet representation (short for RID set representation) is a *data structure* that can hold a set of record IDs (or RIDs); an RSet is a single instance of an RSet representation. Finally, an encoding produces a series of *encoded RSets*, each the union of one or more bins according to some pattern; the purpose of an encoding is to optimize for index size, query performance, etc. Figures 3.1 depicts an example of this process; formal definitions follow.

Definition 3.3.3. A *quantization* Q is a function $Q : V \to K$, where K is a set of *quantized keys*, such that:

1. K has a *total order* $\leq$ such that $Q(v_1) < Q(v_2) \implies v_1 < v_2$ for $v_1, v_2 \in V$ (however, no constraint is made on the ordering of $v_1, v2$ when $Q(v_1) = Q(v_2)$); and
2. Q has a "full inverse" function $Q^{-1} : K \to I_V$ such that $Q(v) = k \iff v \in Q^{-1}(k)$.

$Q^{-1}(k)$ is called the *bin range* of k, and represents the range of values that quantize to k. The upper/lower bounds of the bin range of k are its *bin boundaries*.

Definition 3.3.4. The *quantized bin keys* $K' \subseteq K$ for data variable D under quantization Q are those quantized keys *actually produced* by quantizing D: $K' = \{Q(D(j)) \mid j \in \mathbb{Z}_N\}$. The sorted quantized bin keys of a data variable are denoted as $K' = \{k_1, k_2, \ldots, k_b\}$, where $k_i \leq k_j \iff i \leq j$.

A quantization thus defines a partition of V, and by extension the record IDs $\mathbb{Z}_n$ based on their corresponding values in D. We call this latter partition the *bins* of D, defined as the sequence $B = (B_1, B_2, \ldots, B_b)$ where $B_i = \{j \in \mathbb{Z}_N \mid Q(D(j)) = k_i\}$. Thus, each quantized bin key corresponds to exactly one bin of RIDs. A bin's *bin ID* is its position within B; thus, the bin ID of bin B_i is i.

[1] Note: despite the existence of a quantization step and our use of "bin" terminology, a value-sliced index *need not* use "binning" *per se*; the identity function is a valid quantization, which effectively "disables" binning.

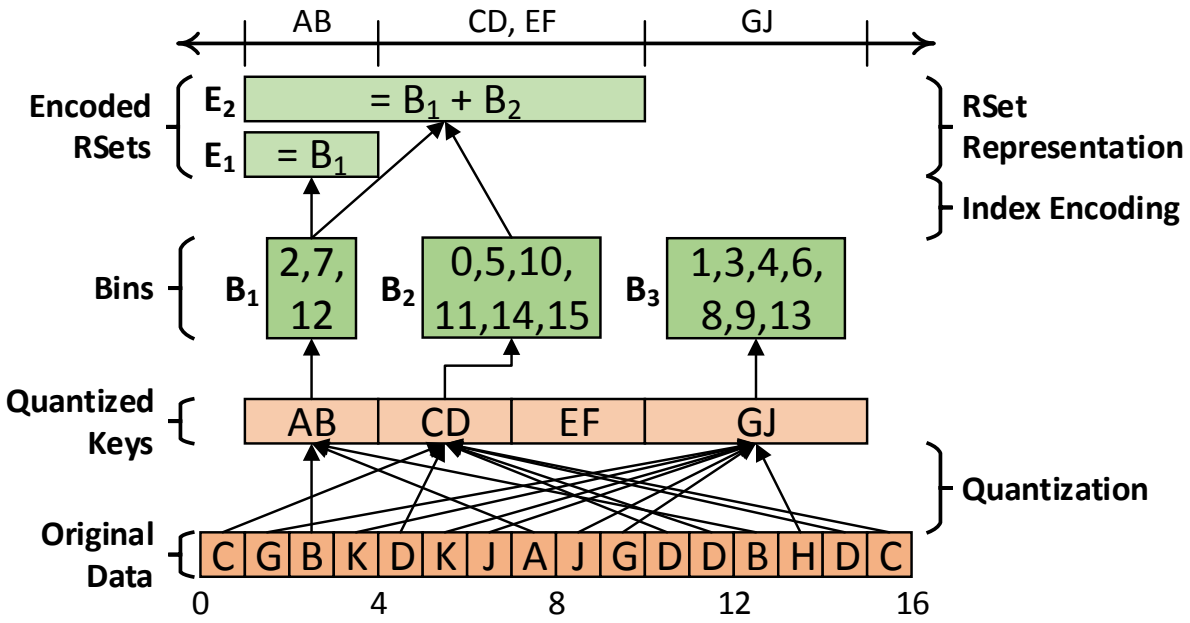


Figure 3.1 An example of the generalized value-sliced indexing model. Indexing relies on three black-box processes: quantization (Definition 3.3.3), RSet representation (Definition 3.3.5), and index encoding (Definition 3.3.6). Key: *A-J* are data values, *0-15* are RIDs, B_x are bins, and E_x are encoded RSets.

Examples of quantizations include *no binning* or *equality binning* ($K = V$ and $Q(v) = v$) and *fixed binning* ($K = \{1, 2, \ldots b\}$, with $i \in K$ corresponding to (non-mutually-overlapping) ranges $v_i = [a_i, b_i) \subset V$, and $Q(v) = i$ s.t. $v \in v_i$). Other examples include "decimal-precision binning" [87] and "significant-bits binning" [42].

Definition 3.3.5. An *RSet representation* R is a bijective function $R : \mathcal{P}(\mathbb{Z}_n) \rightarrow \mathcal{R}$ mapping any set of RIDs to a corresponding *RSet* data structure representing these RIDs ($\mathcal{R}$ is the set of all possible output data structure instances under R). This data structure must support the set operations *union*, *intersection*, and *complement in* $\mathcal{P}(\mathbb{Z}_n)$, and must also be *(de)serializable* from/to binary form; that is, there must exist a bijective function $\mathcal{R} \rightarrow \{0,1\}^*$.

Because R is bijective, there exists inverse function $R^{-1} : \mathcal{R} \rightarrow \mathcal{P}(\mathbb{Z}_n)$ mapping an RSet back to its represented RIDs.

An RSet representation can be thought of as a concrete implementation of the "set of RIDs" abstract data type, while an RSet is an instantiation of some RSet representation. Examples of RSet representations include: *bitmaps* (compressed or otherwise), *inverted indexes* (i.e. RID lists, compressed or otherwise), and the hyperdyadic tree proposed later in this paper.

Definition 3.3.6. An *index encoding* E is a function $E : \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$ mapping a number of bins $b \in \mathbb{N}$ and an encoded RSet ID $i \in \mathbb{N}$ to a set of bin IDs $\beta_i \subseteq \mathbb{N}_b$ indicating how to construct *encoded RSets* for an index with b bins. For a given number of bins b, E defines the number of encoded RSets ρ that will exist, and will return the empty set if an encoded RSet ID outside this range is requested; that is, $E(b,i) \neq \emptyset \iff 1 \leq i \leq \rho$.

When applied in the context of a data variable with bins B and an RSet representation R, the ρ sets of bin IDs $\beta_1, \ldots, \beta_\rho$ returned by the index encoding define ρ corresponding *encoded RSets* $E_1, \ldots, E_\rho$ as $E_i = R\left(\bigcup_{j \in \beta_i} B_j\right)$. That is, the ith encoded RSet is obtained by taking the union of the bins specified by the bin IDs returned by $E(b,i)$, followed by applying the given RSet representation.

Common encodings include *equality encoding*, which stores each bin as its own encoded RSet ($\rho = b$, $\beta_i = \{i\}$), and *range encoding* [16], where each encoded RSet covers bins at or before its position ($\rho = b-1$, $\beta_i = \{1, 2, \ldots, i\}$). Other encodings are mentioned in Section 3.2 and surveyed in prior work [94, 95]. However, discussion of all index encodings in the literature have been limited to the bitmap index up until now; Definition 3.3.6 and later Section 3.3.2 demonstrate for the first time how index encoding may be generalized to *any* RSet representation, not just bitmaps. See Section 3.4 for a detailed analysis of how these existing encodings map to our model.

For an index encoding to be generally useful, it must be "reversible":

Definition 3.3.7. An *index decoder* E^{-1} corresponding to index encoding E is a function $E^{-1} : \mathbb{N} \times \mathcal{P}(\mathbb{N}) \to \mathcal{X}$, whose input a number of bins $b \in \mathbb{N}$ and a set of bin IDs of interest $S \subseteq \mathbb{N}_b$, and whose output is a *mathematical expression* $X \in \mathcal{X}$ over some encoded RSets defined by E combined with set operations (union, intersection, complement) that will compute the union of the bins specified by S. ($\mathcal{X}$ denotes the set of all such mathematical expressions). That is, evaluating $X = E^{-1}(b,S)$ on the actual encoded RSets specified therein will result in a single RSet R_S such that $\texttt{evaluate}(X) = R_S = R\left(\bigcup_{i \in S} B_i\right)$.

In practice, the set of bin IDs S will almost always contain either a single bin, or range of contiguous bins, and good index encodings/decoders will be optimized for these cases. We keep the definition general here, however.

Definition 3.3.8. An index encoding E is said to be *complete*[2] iff a corresponding *index decoder* exists.

Hereafter, we consider only complete index encodings.

Note that an index decoder is defined to return an *unevaluated expression*, rather than an evaluated result. This ensures decoding is possible using only set operations over encoded RSets,

[2] use of "complete" inspired by [17]

with the encoded RSets themselves being treated as "black boxes," thus preserving orthogonality between index encodings and RSet representations.

Finally, we can assemble these the components of quantization, RSet representation, and index encoding to define a value-sliced index:

Definition 3.3.9. A *value-sliced index* $\mathcal{I}_{Q,R,E}$ on a data variable D consists of the list quantized bin keys and the list of encoded RSets yielded by quantization Q, RSet representation R, and index encoding E, as per their individual definitions. That is, $\mathcal{I}_{Q,R,E} = \big((k_1,\ldots,k_b),\big(E_1,\ldots,E_\rho\big)\big)$

As we will show in Section 3.4, the foregoing definitions fully accommodate existing literature on binning, compression, and encoding research, to our knowledge, as well as ALACRITY and the HD-tree of this paper. For example, a simple bitmap index is modeled as (equality binning, bitmap, equality encoding), a common FastBit configuration as (precision binning, WAH bitmap, interval encoding), and ALACRITY as (significant-bits binning, PFOR-Delta inverted list, equality encoding).

3.3.2 Querying a Value-sliced Index

We now show how to compute the query function $\mathcal{Q}$ using a value-sliced index $\mathcal{I}_{Q,R,E}$. The first step in answering a range query is to determine which quantized keys cover the query range. Then, the union of the corresponding bins is computed as a *result RSet*, accomplished by combining some encoded RSets combined via set operations according to the index decoder. Finally, the result RSet may be optionally converted to an RID list for the user.

First, we begin by identifying which index bins are of interest in answering a query:

Definition 3.3.10. Given a query range $q \in I_V$ and a value-sliced index $\mathcal{I}_{Q,R,E}$, a bin $B_i \in B$ with corresponding quantized bin key $k_i \in K'$ is a *touched bin*, and k_i is a *touched bin key*, iff $Q^{-1}(k_i) \cap q \neq \emptyset$.

The set bin IDs for touched bins (or bin keys) under q is denoted as $T(q,K') = \{i \in |K'| \mid k_i \text{ is a touched bin key}\}$.

To compute touched bins in practice, we note that quantized bin keys that fall strictly between the quantized query range endpoints are always touched, and those that fall strictly without are never touched; those equal to a quantized query endpoint require further checking using Definition 3.3.10.

As a special case, we define queries that are "aligned to bin boundaries":

Definition 3.3.11. Given a value-sliced index $\mathcal{I}_{Q,R,E}$ and a query range $q \in I_V$, the query range is said to be *aligned to bin boundaries* or *bin-aligned* iff, for every touched bin B_i with quantized bin key k_i, $q \cap Q^{-1}(k_i) = Q^{-1}(k_i)$, that is, the bin range of every touched bin is covered by q.

For now, we assume all query ranges are aligned to bin boundaries. In this case, the set of RIDs whose values fulfill a query q is exactly the union of all touched bins, that is, $\mathcal{Q}(q) = \bigcup\{B_i \mid i \in T(q, B)\}$. Extension to non-aligned queries via *candidate checks* is discussed in Section 3.3.2.

In order to recover the union of the touched bins, it is necessary to decode the *index encoding* that was applied. This is relatively straightforward given the existing of an index decoder E^{-1}; therefore, we can now define an algorithm QUERY that computes $\mathcal{Q}$:

function QUERY($q \in I_V, \mathcal{I}_{Q,R,E}$)
  $S = T(q, K')$ ▷ compute touched bin IDs
  $X = E^{-1}(b, S)$ ▷ compute expression X for $R\left(\bigcup_{i \in S} B_i\right)$
  $R_S = \texttt{evaluate}\left(X, \mathcal{I}_{Q,R,E}\right)$ ▷ evaluate X over the index's encoded RSets to get result RSet
  $A = R^{-1}(R_S)$ ▷ convert result RSet to RIDs (optional)
  return A ▷ return answer to query
end function

An interesting point is that, while the most straightforward way to evaluate X is to utilize the RSet representation's set operation algorithms to evaluate the expression directly, this algorithm does not dictate a specific approach, and so any method may be used to compute $\texttt{evaluate}(X)$. This may include specialized methods for specific quantizations, RSet representations, and/or index encodings. For example, in the case of equality encoding, X is simply the union of some number of bin RSets. In this case, it is permissible to evaluate X by taking the union of the bin RSets *not* present in X, followed by taking the complement. This approach may be more efficient when a majority of bins are touched by a given query. More radical query evaluation approaches are also possible, even producing a result RSet of a completely different RSet representation from that of the encoded RSets; see future work in Section 6.2.4 for an extended discussion.

3.3.3 Multi-variate Queries

Having established how to evaluate a query over a single data variable, extension to multi-variable queries is straightforward, using the same approach as used in bitmap indexing. First, the constraints on individual variables are evaluated independently, using each variable's own index. However, instead of ending by converting the resultant RSets back to RIDs immediately, the RSets for the constraints are instead combined using further set operations corresponding to logical operations specified in the query. For example, given a hypothetical query constraint "*temperature* > 30 AND *windspeed* > 80 OR *pressure* < 1000", each of the three constraints would first be evaluated independently. Then, the *temperature* RSet would be combined via intersection with the *windspeed* RSet, after which the resulting RSet would be combined with the *pressure* RSet via union.

After the multi-variate constraint is fully evaluated, the final RSet is then converted back to RIDs and returned to the user, as in the single variable case.

3.3.4 Candidate Checks and Non-aligned Queries

Up to now, we have assumed all queries are bin-aligned. In practice, however, this is not always the case; when (non-equality) binning is used, most queries will incur *candidate bins* that are only partially covered by the query. For exact query results, original values at all candidate RIDs must be compared against the query range, in a process known as *candidate checking* [82]. Though we do not delve into candidate checks in this work, we believe the core definitions here are a sufficient foundation for future extensions, including candidate checks.

3.4 Modeling Previous Work

Having developed a generalized indexing model, we now apply it to a variety of prior work. Besides demonstrating the flexibility of the model, we intend this section as a concise summary of many existing techniques in the literature.

3.4.1 Quantizations (Binning Methods)

Definition 3.3.3 states that every quantization must specify three things:

- K, the set of possible quantized keys;
- $Q(v)$, the quantization function; and
- $Q^{-1}(k)$, the full inverse quantization function.

We give concrete specifications for some common binning techniques from the literature below.

3.4.1.1 Identity Binning

The simplest quantization method is to perform no value mapping at all, i.e., the identify function:

$$\begin{aligned} K &= V \\ Q(v) &= v \\ Q^{-1}(v) &= v \end{aligned}$$

3.4.1.2 Equal-width Binning

Equal-width binning is likely the next simplest quantization. The domain is partitioned into (half-open) intervals of width $w \in V, w > 0$ and with phase $p \in V, 0 <= p < w$. These intervals take on the form $\left[i\,w + p, (i+1)w + p \right)$ for $i \in \mathbb{Z}$, and each becomes a bin range:

$$\begin{aligned} K &= \mathbb{Z} \\ Q(v) &= \left\lfloor \frac{v-p}{w} \right\rfloor \\ Q^{-1}(k) &= \left[k\,w + p, (k+1)w + p \right) \end{aligned}$$

3.4.1.3 FastBit Decimal-precision Binning [87]

Precision-d binning rounds a real-valued input to $d \in \mathbb{N}$ "significant digits" (i.e., d decimal digits in the significand when represented in scientific notation) and bins together those values that are equal after this rounding.

In the following equations, $M(x)$ denotes the "order of magnitude" of x, i.e., $M(x) = \lfloor \log_{10} x \rfloor$.

$$\begin{aligned} K &= \mathbb{R} \\ Q(v) &= \text{round } v \text{ to } d \text{ significant digits} \\ &= \left\lfloor v \cdot 10^{-(M(v)-(d-1))} + 0.5 \right\rfloor \cdot 10^{M(x)-(d-1)} \\ Q^{-1}(k) &= \left[k \pm 0.5 \cdot 10^{M(k)-(d-1)} \right) \end{aligned}$$

3.4.1.4 ALACRITY Significant-bits Binning [41]

Significant-bits binning uses the first b bits in the binary representation of floating-point values as the quantized key.

This form of binning relies on a key property of the IEEE 754 [39] binary floating-point format, which represents a real value with the following three bit fields packed into a single word, listed from high-order to low-order: 1) a single sign bit, 2) exponent bits, and 3) mantissa (or significand) bits. Specifically, floating-point values under this standard have the property that, if their binary representations are interpreted as a *ones'-complement integers*, their total order is still maintained. That is, if $C(x)$ reinterprets floating-point value x as a ones' complement integer, then $x < y \iff C(x) < C(y)$. Even $\pm\infty$ and subnormal values obey this rule, though of course NaN values do not.

Two corollary properties follow from this basic principle. First, all floating-point values sharing a common *prefix* of bits lie in a single, contiguous range. Second, comparing the prefix of two

floating-point values with different prefixes has the same result as comparing the values themselves. These two results indicate that a floating-point bit prefix is a viable quantization method.

In practice, whenever b is large enough to cover the sign bit and all of the exponent bits (and also possibly some mantissa), significant-bits binning is effectively *base-2 precision binning*. That is, it has the effect of rounding values represented in base-2 exponential notation to a certain number of significant (binary) figures. Thus, it behaves similarly to decimal-precision binning, with the advantage of being simpler and more numerically-stable to compute and the drawback of not mapping as cleanly to scientific practice.

Significant-bits binning is defined below. We use the following notation:

- t is the number of bits in a floating-point word (e.g., 32 for a single-precision floating point such as "`float`" in C)
- Function $\mathrm{C}(x)$ represents reinterpreting floating-point value x as a ones' complement integer (with inverse $\mathrm{C}^{-1}(x)$)
- $\mathrm{B}(k)$ represents a bitmask with the lowest k bits set and all other bits clear
- Bitwise OR is represented by $|$
- (Unsigned) left and right shift are represented by $\ll$ and $\gg$, respectively

$$
\begin{aligned}
K &= \text{the set of ones' complement integers with } b \text{ bits} \\
Q(v) &= \mathrm{C}(v) \gg (t-b) \\
Q^{-1}(k) &= \left[\mathrm{C}^{-1}(k \ll (t-b)), \mathrm{C}^{-1}(k \ll (t-b)) \mid \mathrm{B}(t-b)\right)
\end{aligned}
$$

3.4.2 RSet Representations

The two key properties of a RSet representation as given by Definition 3.3.5 are as follows:

- support for *set operations*, specifically *union*, *intersection*, and *complement*; and
- support for (de)serialization to binary form suitable for storage.

We summarize how several indexing approaches meet these criteria in Table 3.1.

3.4.3 Index Encodings

Recall that a (complete) encoding consists of two pieces. First, an *encoding* E must be specified, which determines the quantity ρ and composition $\beta_1, \ldots, \beta_\rho$ of encoded RSets based on the number of bin RSets b (Definition 3.3.6). Second, an index decoder E^{-1} must exist to reverse the encoding (Definition 3.3.7). We now show how these definitions are fulfilled by common encodings from the literature, which previously were only conceived of in the context of bitmap indexing, but which have now generalized to any value-sliced index.

For brevity, we make the following simplification in our definitions. While a general index decoder E^{-1} is defined to map an arbitrary set S of bin IDs to an encoded RSet expression, all of the following encodings are optimized for *equality and/or range queries*, in which S consists of one or more contiguous bin IDs. Thus, we give formulae and algorithms only for the case $E^{-1}(b, S = \{x, x+1, \ldots, y\})$, which we denote as $E^{-1}(b, x, y)$. The full index decoder for more general S can then be realized by returning the union of the expressions for each contiguous bin range in S.

In what follows, we use **0** to represent the empty RSet (containing no RIDs) and **1** to represent the universal RSet (containing all RIDs). Also, we use $\overline{A}$ to denote set complement of a RSet A, and $A \setminus B$ for set difference betweeen RSets A and B (equivalent to $A \cap \overline{B}$).

3.4.3.1 Equality Encoding [5, 61]

The most straightforward index encoding is to simply use the bin RSets as-is; this is known as "equality encoding" (or simply no encoding).

$$\begin{aligned} \rho &= b \\ E(b, i) &= \{i\} \\ E^{-1}(b, x, y) &= \text{expression}\left[E_x \cup E_{x+1} \cup \cdots \cup E_y\right] \end{aligned}$$

Table 3.1 A table of existing RSet representations, as interpreted within our formal model and demonstrating how the model requirements are fulfilled.

RSet representation	**Set operations**	**Serialize as...**
Uncompressed bitmap	Bitwise operations	Simple bitmap
WAH-compressed bitmap	WAH bitwise operations	Array of WAH words
Inverted list	List merge, intersection, etc.	Integer array
Compressed inverted list	Decompress, then list merge, etc.	Compressed chunks
kHD-tree	See Sections 2.2.4, 2.3.3	Array of kHD-tree words

3.4.3.2 Range Encoding

Range encoding employs one encoded RSet per bin RSet (except the last), where each encoded RSet contains all RIDs from bins less than or equal to its corresponding bin No encoded RSet is stored for the last bin, since it would simply be the universal RSet **1**.

$$\rho = b-1$$
$$E(b,i) = \{1,2,\ldots,i\}$$
$$E^{-1}(b,x,y) = \begin{cases} \text{expression}[\mathbf{1}] & \text{if } x=1, y=b \\ \text{expression}\left[E_y\right] & \text{if } x=1, y<b \\ \text{expression}\left[\overline{E_{x-1}}\right] & \text{if } x>1, y=b \\ \text{expression}\left[E_y \setminus E_{x-1}\right] & \text{otherwise} \end{cases}$$

3.4.3.3 Interval Encoding [17]

Interval encoding is similar to range encoding in that each encoded RSet covers a contiguous range of bin RSets, and bins RSets are recovered by combining endpoint encoded RSets. In contrast to range encoding, however, interval encoding uses about half as many encoded RSets, all encoded RSets cover the same number of bins, and the decoding process is substantially more complex.

In what follows, let $w = \left\lfloor \frac{b}{2} \right\rfloor$ (this stands for the "interval width").

$$\rho = \left\lceil \frac{b}{2} \right\rceil$$
$$E(b,i) = \{i, i+1, \ldots, i+w-1\}$$
$$E^{-1}(b,x,y) = \begin{cases} \text{expression}[\mathbf{1}] & \text{if } x=1 \wedge y=b \\ \overline{E^{-1}(b,1,x-1)} & \text{otherwise, if } y=b \\ \text{expression}[E_x] & \text{otherwise, if } y-x=w-1 \\ \text{expression}\left[E_x \cup E_{y-(w-1)}\right] & \text{otherwise, if } y-x>w-1 \\ \text{expression}\left[E_x \setminus E_{y+1}\right] & \text{otherwise, if } y<w \\ \text{expression}\left[E_{y-(w-1)} \setminus E_{x-w}\right] & \text{otherwise, if } x>w \\ \text{expression}\left[E_{y-(w-1)} \cap E_x\right] & \text{otherwise} \end{cases}$$

3.4.3.4 Multi-level Encoding [94, 95]

Multi-level encoding is a strategy for including multiple, independent groups of encoded RSets in the same index (these groups are called *levels*). Any number of levels $\ell > 1$ greater than one is possible, though prior work indicates that two levels is optimal [94, 95]. Each level i is assigned its own *inner encoding* $\mathcal{E}_i$, selected from existing techniques (such as equality, interval, etc.). A multi-level encoding is typically named by listing its inner encodings from highest level to lowest; for example, an *interval-equality* encoding uses interval encoding on the second-level and equality on the first.

The multi-level encoding begins by applying its first-level inner encoding $\mathcal{E}_1$ to the full set of bin RSets to produce the first-level set of encoded RSets $E_{1,1}, E_{1,2}, \ldots, E_{1,r_1}$; this step proceeds exactly as if the first-level inner encoding were the overall encoding used. Then, for each successive level $i > 1$, some *bin coarsening function* C partitions the all bins into contiguous groups, which are merged into a set of b_i *coarse bins* denoted as $B_{i,1}, B_{i,2}, \ldots, B_{i,b_i}$ (for consistency, we define $b_1 = b$ and $B_{1,j} = B_j$). More formally, the bin coarsening function $C : \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$ takes as input a level number $i \in \mathbb{N}_\ell$ and a coarse bin ID at level $j \in \mathbb{N}_{b_i}$ and returns the set of contiguous (original, non-coarse) bin IDs it consists of: if $C(i, j) = S$, then $B_{i,j} = \bigcup_{k \in S} B_k$. These coarse bins are required to be *nested* within any higher levels: every bin at level i must be either fully contained in or fully disjoint from every coarse bin at every higher level. A simple example would be to merge bins in groups of some fixed size c; in this case, a b-bin dataset would yield $b_2 = \frac{b}{c}$ coarse bins at the second level, $b_3 = \frac{b}{c^2}$ at the third level, and so on; in this case, $C(i, j) = ((j-1)c^i + 1, jc^i)$.

Once all levels of bins have been determined, each level is then encoded separately using the level-i inner encoding $\mathcal{E}_i$. This produces level-wise encoded RSets $E_{i,1}, E_{i,2}, \ldots, E_{i,\rho_i}$, which are collected to form the full set of encoded RSets.

While each level is, by itself, a complete encoding (i.e., is capable of answering any query), the strength of the multi-level encoding is how the decoder algorithm can coordinate the levels to speed up query processing across a variety of query types. For example, consider an interval-equality encoding. Equality queries involving a single bin may be quickly answered by consulting the equality-encoded first level, as the original bin RSets are directly accessible.

For a range query touching many bins, on the other hand, the second-level interval encoding may be consulted to cover a large portion of the range with just two encoded RSets. The second level is built over coarser-grained bins, however, so it is likely the query range cannot be exactly matched; instead, the finer-grained first-level bins are used to "round out" the edges of the query range. This strategy reads far fewer encoded RSets than a purely equality-encoded index, due to the efficiency of interval encoding at computing large ranges of bin RSets with at most two encoded RSets. At the

same time, the index is far smaller than a purely interval-encoded index, assuming a compressed RSet representation is used, as the coarseness of the second level greatly reduces the number of interval-encoded RSets, which are storage-heavy due to the ineffectiveness of RSet compression under interval encoding [92].

$$\rho = \sum_{i=1}^{\ell} \rho_i$$

$$\rho_i = \text{the number of inner encoded RSets determined by } \mathcal{E}_i \text{ as if for } b_i \text{ bins}$$

$$E(b,j) = \begin{cases} E'(1,j) & \text{if } 1 \le j \le \rho_1 \\ E'(2, j-\rho_1) & \text{if } \rho_1 < j \le \rho_1 + \rho_2 \\ \cdots & \\ E'\left(\ell, j - \sum_{i=1}^{\ell-1} \rho_i\right) & \text{if } \sum_{i=1}^{\ell-1} \rho_i < j \le \rho \end{cases}$$

$$E'(i,j) = \text{the result of applying } \mathcal{E}_i \text{ on coarse bin } j\text{, then converting to original bin IDs}$$

$$= \bigcup_{j_{\text{coarse}} \in \mathcal{E}_i(b_i, j)} C(i, j_{\text{coarse}})$$

An index decoder algorithm for multi-level encoding is given in Algorithm 3. More efficient implementations are surely possible, but our intention is only to demonstrate the nature of an index decoder algorithm.

3.4.3.5 Multi-component Encoding [16]

Multi-component encoding is related to multi-level encoding, in that it forms a composite of multiple simpler encodings, but it considerably more complex. We make our best attempt to summarize the key principles of this method here, but to avoid veering too far off topic (demonstrating the comprehensiveness of our indexing model), the explanation is brief and some details are omitted. It is suggested that the reader consult the original papers [16, 17] and/or later surveys [94, 95] for a more thorough explanation, though it may be challenging to map the concepts presented therein to this more general treatment.

Multi-component encoding is parameterized by a sequence of m integers $c_1, c_2, \ldots, c_m$ for some m with $c_i > 1$ and $\prod_{i=1}^{m} c_i \ge b$; this sequence is called the *base*. When all c_i share a common value c, we say the base is *uniform* and call this a *base-c* multi-component encoding. For any integer $0 \le x < b$, we define *base representation* of x in a given base as the sequence of digits $d_1, d_2, \ldots, d_m$

Algorithm 3 Index decoder algorithm for general multi-level encoding

In: x, y: the inclusive bounds the bin range to compute, with $1 \le x \le y \le b$
Out: X: an expression over E_i to compute the bin range RSet

```
result ← empty expression
remainingBins ← {x, ..., y}
for i in ℓ, ℓ−1, ..., 1 do
    ▷ find coarse bins at this level covering some of the bin range, without going outside it
    levelBins ← {}
    for j in 1, ..., b_i do
        binContents ← C(i, j)
        if binContents ⊆ remainingBins then        ▷ if coarse bin j covers some of the bin range
            Remove binContents from remainingBins
            Add j to levelBins
        end if
    end for
    ▷ obtain the subexpression to decode these coarse bins from this level
    subResult ← E_i^{-1}(b_i, levelBins)
    ▷ add the subexpression to the final result expression
    Shift encoded RSets referenced in subResult, adding Σ_{k=1}^{i-1} ρ_i to each subscript
    result ← expression[result ∪ subResult]
end for
Assert remainingBins = {}
return result
```

such that $x = \sum_{i=1}^{m} d_i \prod_{j=1}^{i-1} c_j$ and $\forall i : 0 \leq d_i < c_i$.

A multi-component encoded index consists of *m components*. Each component i is, itself, a "mini-index" over a set of *projected bins* $B_{i,1}, \ldots, B_{i,c_i}$, where $B_{i,j}$ is the union of all bins B_x where $d_i + 1 = j$ in x's base representation. These projected bins are then encoded with some *inner encoding* $\mathcal{E}_i$ producing a group of *inner encoded RSets* $\mathcal{E}_{i,1}, \ldots, \mathcal{E}_{i,\rho_i}$ (in the common case of equality encoding, these are just $B_{i,1}, \ldots, B_{i,c_i}$). Once all components are complete, the overall encoded RSet list is simply the concatenation of the inner encoded RSet lists: $\mathcal{E}_{1,1}, \ldots, \mathcal{E}_{1,r_1}, \mathcal{E}_{2,1}, \ldots, \mathcal{E}_{2,r_2}, \ldots, \mathcal{E}_{m,\rho_m}$.

$$\rho = \sum_{i=1}^{m} \rho_i$$

$\rho_i =$ the number of inner encoded RSets determined by $\mathcal{E}_i$ as if for c_i bins

$$E(b,j) = \begin{cases} E'(1,j) & \text{if } 1 \leq j \leq \rho_1 \\ E'(2, j-\rho_1) & \text{if } \rho_1 < j \leq \rho_1 + \rho_2 \\ \cdots & \\ E'\left(\ell, j - \sum_{i=1}^{\ell-1} \rho_i\right) & \text{if } \sum_{i=1}^{\ell-1} \rho_i < j \leq \rho \end{cases}$$

$E'(i,j) = \{x \mid d_i = j \text{ in the base representation of } x\}$

Algorithm 4 gives a decoding algorithm to handle one-sided bin ranges (i.e., of the form $[1 \,..\, x]$) for multi-component encoding. Two-sided range support is a trivial extension effected by taking the set difference of the upper and lower ranges, with some special cases.

This algorithm is generalized to support any base and combination of inner encodings. More efficient versions are certainly possible for special cases, however; for instance, components with range encoding can be optimized further, as shown in Figure 6 of the original paper [16]. Interestingly, some of these optimizations might possibly be automated with a good symbolic solver, since the decoding algorithm builds an unevaluated expression that would be amenable to such an approach; we leave potential investigation of this to future work.

3.4.3.6 Binary Encoding [60, 85]

Binary encoding (also known as bit-sliced indexing) is a special case of multi-component encoding, specifically base-2 component encoding [16]. Therefore, the above definitions and algorithm apply.

Algorithm 4 Index decoder algorithm for recovering a one-sided range of bins in a general multi-component encoding

In: u: the inclusive upper bound of the one-sided bin range to compute, with $1 \leq u < b$
Out: X: an expression over E_i to compute the bin range RSet
▷ NB: base representation digits d_i are zero-based, while bin IDs are one-based
Compute base representation of u as $d_1, \ldots, d_m$
$result \leftarrow \mathcal{E}_1^{-1}(1, d_1 + 1)$ ▷ expression for union of projected bins $B_{1,1}, \ldots, B_{1,d_1+1}$
for i in $2, \ldots, m$ **do**
  $limiter \leftarrow \mathcal{E}_1^{-1}(d_i + 1, d_i + 1)$ ▷ expression for projected bin B_{i,d_i+1}
  Shift encoded RSets referenced in *limiter*, adding $\sum_{k=1}^{i-1} \rho_i$ to each subscript
  $result \leftarrow$ expression$[result \cap limiter]$
  if $d_i > 0$ **then**
    $leftexpr \leftarrow \mathcal{E}_1^{-1}(1, (d_i - 1) + 1)$ ▷ expression for union of projected bins $Bi, 1, \ldots, B_{i,d_i}$
    Shift encoded RSets referenced in *leftexpr*, adding $\sum_{k=1}^{i-1} \rho_i$ to each subscript
    $result \leftarrow$ expression$\big[result \cap leftexpr\big]$
  end if
  return *result*
end for

3.4.4 Value Domains

Finally, we also want to briefly address the nature of variable value domains. Our model is intended to be general across many types of data. Numerical data, especially real-valued data, is the most common in scientific computing, and is likely the easiest case to reason about. Nonetheless, the value domain V may be any totally-ordered set. For example, the set of strings over some alphabet (e.g., ASCII, Unicode, etc.) under lexicographic order is a valid value domain in this model, and would be useful in an information retrieval context.

We also note that a stripped-down version of the model would be applicable even for domains without a total order. Such a system could be derived from the total-order version by imposing an arbitrary total order on the unordered (or partially-ordered) domain, then restricting to the use of identity quantization, equality index encoding, and equality queries only (any RSet representation would be acceptable, though). Such a configuration could be useful in the case of, e.g., unordered categorical data.

3.5 Indexing with PIQUE

We have now defined a generalized indexing and query model, which encompasses both existing bitmap and ALACRITY indexes as well as our new HD-tree index described in Chapter 2. Our next step is to realize the insight granted by this model by building an implementation of a general indexing and query platform. To this end, in this section, we present the Parallel Indexing and Query Unified Engine (or PIQUE for short). PIQUE's generic indexing algorithm is described in Section 3.5.1, and the generic query algorithm is discussed in Section 3.5.2 below.

PIQUE's modularity enables the implementation of a wide range of quantizations, RSet representations, and index encodings; the following are currently supported:

- **Quantizations:** user-specified bin boundaries, "precision" binning (from FastBit [87]), "significant-bits" binning (from ALACRITY [42])
- **RSet representations:** *k*HD-tree, uncompressed bitmaps, WAH-compressed bitmaps (from FastBit [89]), both PFOR-Delta-compressed and uncompressed inverted lists (from ALACRITY [40])
- **Encodings:** equality, range, interval, binary [16, 17, 61, 94, 98]

Adding new implementations is straightforward. PIQUE is written predominantly in in C++, and each category above provides a clean object-oriented interface to admit new algorithms/data structures as drop-in replacements. For instance, there is a set of classes for precision quantization, another set for WAH bitmaps, and a third for interval encoding.

3.5.1 Indexing

The core of the indexing algorithm relies on "RSet builder" concept developed in Section 2.3.2. Following this model, each RSet representation must provide an associated stateful "incremental builder" (denoted as `Builder` in the algorithm), representing a partially-built RSet and with two member functions: `push_run`($length, is\_filled$) (for pushing a run of $length$ present/absent RIDs, depending on is_filled), and and `getFinalRSet`(), to convert to a true, completed RSet. As discussed in Section 2.3.2, it is always possible to write a builder for a given RSet representation.

Indexing proceeds in two phases. First, the algorithm scans the dataset, quantizing the data and identifying runs of equal quantized keys. These runs are added to RSet builders via the `push_run` interface, with a new RSet builder being instantiated whenever a new quantized key is seen. At the end of this scan, the RSet builders are sorted by quantized key and converted to completed *bin RSets*. We use RSets to store bin RIDs, rather than some other data structure, because RSets are the

natural output from builders, are assumedly space-efficient, and are conducive for index encoding in the next phase.

During the second phase, the chosen index encoding is consulted, and the bin RSets are composed into the final *encoded RSets* according to the pattern returned. As an optimization, some encodings have a specialized routines. For instance, under range encoding, the ith encoded RSet (E_i) is the union of bin RSets 1 through i ($R(B_1), \ldots, R(B_i)$). Instead of computing this directly, we use $E_i = E_{i-1} \cup R(B_i)$, reusing previous encoded RSets to reduce union operations from $\frac{b(b-1)}{2}$ to $b-1$.

Algorithm 5 The generic indexing algorithm

In: D: an array of values to index (the dataset)
In: Q, R, E: quantization, RSet representation, encoding
Out: $\mathcal{I}_{Q,R,E}$: an index over D

▷ Scan the dataset, build bin RSets
$qtor \leftarrow$ new map ▷ map quant. key to RSet builder
$i \leftarrow 0$
while $i < |D|$ **do**
  $q \leftarrow Q(D[i])$
  $j \leftarrow$ first $j > i$ with $Q(D[j]) \neq q$ (or $|D|$ if $\nexists$ such j)
  if $qtor[q]$ is null **then** $qtor[q] \leftarrow$ `new Builder(`R`)`
  $qtor[q]$`.push_run(`$i - qtor[q].pos$`,false)`
  $qtor[q]$`.push_run(`$j - i$`,true)`
  $i \leftarrow qtor[q].pos \leftarrow j$
end while
▷ Finalize and collect bin RSets sorted by quant. key
$keys, bins \leftarrow$ new arrays
for $(q, b) \in qtor$ (visit in key-sorted order) **do**
  b`.push_run(`$|D| - b.pos$`,false)` ▷ "top off" builder
  $keys$`.append(`q`)`
  $bins$`.append(`b`.getFinalRSet())`
end for
▷ Compose bin RSets into encoded RSets
$binsPerRSet \leftarrow E(|keys|)$
$rsets \leftarrow$ new array
for $rsetBins \in binsPerRSet$ **do**
  $rsets$.append$\left(\bigcup_{b \in rsetBins} bins[b]\right)$
end for
return new Index(Q, R, E, keys, rsets)

3.5.2 Query Processing

Unlike the indexing algorithm, which entails several practical considerations vs. the theoretical definition, query processing for a single variable is fairly straightforward, and is implemented essentially as presented earlier in Definition 3.3.2:

1. find the range of touched bin IDs based on the query,
2. use the index decoder to determine which RSets are needed to compute the union of these bins,
3. read those encoded RSets from disk, and
4. evaluate the expression returned by the index decoder using the read RSets.

PIQUE also supports multi-variate queries, consisting of one or more equality constraints (e.g., $temperature = 300$) and/or range constraints (e.g., $100 < temperature < 200$) combined with standard boolean operators (AND, OR, NOT). In such a multi-variate query, all constituent single-variable constraints are first evaluated using the above procedure. However, instead of returning RID lists, these queries are set to return RSets without conversion. Then, an expression tree is formed, and these intermediate RSets are combined via set operations (union for OR, intersection for AND, complement for NOT). The final RSet returned at the root is then converted to an RID list for the user.

Currently, candidate checks are not yet implemented for single-variable constraints; all queries are answered solely from the index. However, as discussed in Section 3.3.4, it is feasible to implement candidate checks under the generalized query processing model.

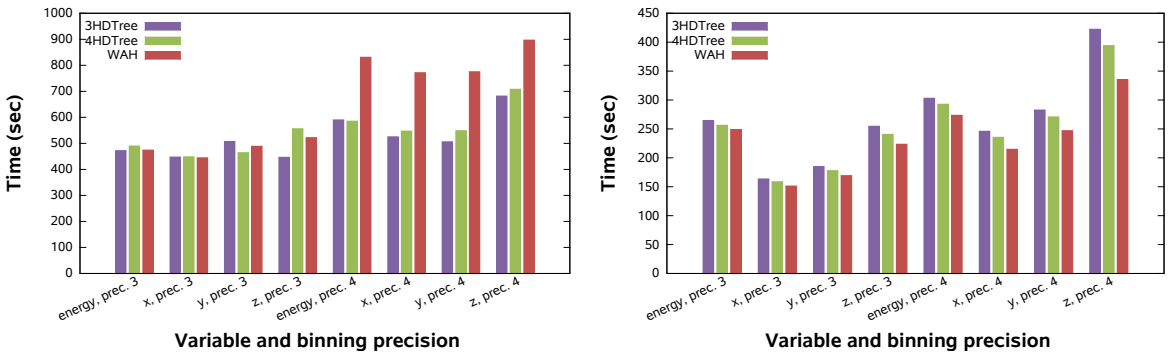
3.5.3 Parallelization

The common way to parallelize indexing and query is to divide the input data into chunks ("partitions"), then index/query each independently. Traditional DBMSs term this technique "horizontal partitioning," and it is also used by FastQuery [23] (parallel FastBit) and DIRAQ [48] (parallel ALACRITY). We adopt this approach in PIQUE.

Even still, several approaches exist to build a partitioned index in parallel. In PIQUE, we opt for a single, shared index file (mainly to simplify query processing), so file synchronization is a concern. The most straightforward way to regulate shared index file access is to pre-allocate input data partitions across cores, then use MPI collective I/O to maintain file synchronization. At each step, each rank would read its next allocated partition, index it, exchange index size information globally, then participate in a collective write to file. However, this approach is susceptible to stragglers: between collective writes, progress is made only at the speed of the slowest core.

In PIQUE, we loosen synchronization in the following way. First, a single core is designated as the "controller." Each rank still operates on pre-allocated partitions, but when a core finishes indexing a partition, it messages the controller with that index partition's length. In response, the controller returns the next available file offset, incrementing the current offset by the partition's length. On receiving a file offset, the indexing core can write its index partition and immediately proceed to the next input partition. In this way, faster cores may proceed while slow cores are still working. The straggler problem is not entirely eliminated, however: each core still has a fixed workload. Work stealing could be used to mitigate this issue.

Parallel querying may be accomplished via a similar pattern: each core independently reads index partitions, decoding each one, with results being collected to a master process or file on disk. At this time, the parallel query implementation in PIQUE is still a work-in-progress. Due to our aforementioned embarrasingly-parallel approach to query parallelization, we see no reason why the generality of PIQUE would be compromised. Moreover, because query processing is inherently far faster than index building, PIQUE achieves good performance in our experiments even in serial.



(a) Index build time (total)

(b) Index build time (CPU-only portion)

Figure 3.2 Index build time, total (including I/O) and CPU-only, over different variables, binning methods, and RSet representations. In all cases, the indexing was performed in parallel on 256 cores using PIQUE. Build times are similar between WAH and HD-trees, with somewhat-higher CPU times for HD-trees balanced by reduced I/O.

3.6 Results

The central contribution of our abstract indexing and query model are to conceptually unify existing research in this space; we demonstrate this in Section 3.4 as a sort of "theoretical results" section. In

contrast, the main result of PIQUE is its delivery of concrete implementations of many indexing and query techniques on one platform, and in corollary, the comparisons between indexes enabled by this. Therefore, our results here focus on further evaluation of HD-trees and WAH, using PIQUE as a medium.

Nevertheless, it is also possible to evaluate shared core algorithms of PIQUE. Therefore, we also conduct a scalability evaluation of the generalized parallel indexing routine in PIQUE, comparing the PIQUE WAH implementation with the existing parallel implementation of FastBit, called FastQuery [23].

3.6.1 Experimental Setup

Our experimental setup is largely the same as in Chapter 2 (see Section 2.4.1); however, we repeat the relevant information here, with some additions in light of the parallel experiments.

All of our experiments were run on the Edison supercomputer at the National Energy Research and Scientific Computing Center (NERSC) [56]. Each node contains 24 cores with 64 GB shared memory. Edison's Lustre parallel filesystems were used for storage in each experiment, specifically the "/scratch1" and "/scratch2" filesystems, each consisting of 96 Lustre OSTs providing a total of 2.1 PB of storage and 48 GB/s aggregate peak I/O performance [26, 57] (we used two filesystems due to storage quota limitations, but each experiment was performed entirely on one or the other).

We run our evaluations using a large-scale dataset from VPIC [9–11], a three-dimensional electromagnetic relativistic kinetic plasma simulation code, consisting of 7 single-precision floating-point variables: energy, x/y/z (position), and Ux/Uy/Uz (velocity components relative to the underlying magnetic field). However, we only use the first four in our experiments. Additionally, based on precedent [14], we use a pre-filtered subset consisting of "interesting" particles, defined by $energy \geq 1.1$, yielding 180 billion particles and a 5 TB dataset ($\tilde{7}$50 GB per variable).

Since the data are floating point, it is necessary to employ binning. We use decimal-precision binning as developed and used in FastBit [87], in order to ensure comparability with our system. Under precision-d binning, values are binned together when equal after truncating at d digits of (decimal) precision (i.e., d significant digits). For example, the values 1.31×10^3 and 1.34×10^3 would be binned together under precision-2, but not under precision-3 or higher.

For software, we use the FastQuery [23] indexing and query system, which is a layer on top of FastBit [88, 91] (developed by the same organization) that provides additional features, which will be useful for comparison in later chapters. Since they are so closely related, we use the terms FastBit and FastQuery interchangeably. FastQuery version 0.8.2.8 and FastBit version 1.3.9 were used, the latest versions available at the time we began our experiments. However, we do fix one important

bug in FastBit 1.3.9: in our early experiments, FastBit was loading the entire index into memory for each variable queried, even when only a small fraction was needed. We have reported and confirmed this issue with the developers; correcting this bug significantly improved I/O times for FastBit (and FastQuery) in our tests.

3.6.2 Indexing Performance

Now that we have discussed the parallel indexing implementation in PIQUE, we return to the earlier experiment in Section 2.4.2 and examine index build times across multiple RSet representations (HD-tree and WAH), data variables, and binning configurations.

Using PIQUE, we again measure indexing performance under the 3HD-tree, 4HD-tree, and WAH RSet representations on four VPIC variables. We repeat the experiment once each using precision-3 and precision-4 binning, for a total of eight variable/binning cases. As a reminder, precision-d binning groups values equal to d significant (decimal) digits, and is ported from FastBit [87] to PIQUE for cross-system comparability in later experiments. All indexes are built in parallel using 256 cores.

Figure 3.2 shows the comparison of index build times across all of these experiments. We see that all RSet representations yield similar timings within the same variable/bin method. CPU time to build HD-tree indexes is somewhat higher (Figure 3.2b), which is not surprising given the relative complexity of the "HD-tree builder" algorithm described in Section 2.3.2. However, this increased compute time is offset by the benefit of a reduced index size (and therefore lower write cost) relative to the WAH index (as covered back in Section 2.4.2 in Figure 2.6). These results support the conclusion that, despite offering higher index compression, HD-tree indexing is not more expensive to perform than WAH. Note: the time spikes for WAH on decimal-precision 4 binning appear to be primarily due to I/O variability; however, it is plausible that HD-tree indexing may be less susceptible to this kind of randomness due to reduced overall I/O.

As determined with index size in Section 2.4.2, indexing time also varies with the variable and binning method used, regardless of RSet representation, and is strongly correlated with the *number of bins* produced in the index. The binning method is partly responsible: for all variables, precision-4 binning yields 7-9x as many bins as precision-3, explaining the higher storage cost in these cases. The remainder of this effect comes from the variables themselves; as previously discussed, the variable z yields over 10x more bins than any other variable (under either binning method) due to its value distribution and the nature of the exponential-based precision binning.

To conclude, we infer that, while the HD-tree RSet representation requires less storage space than WAH as previously determined, it nonetheless has consistently similar total index build time

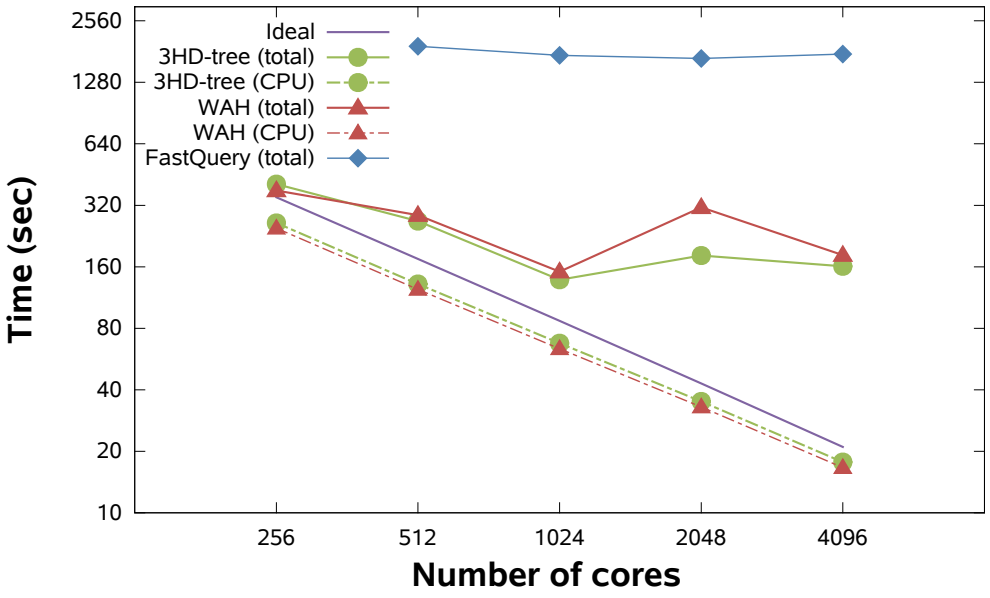


Figure 3.3 Strong scalability of parallel indexing, with total time and CPU-only time (i.e., I/O excluded) shown. The CPU portion is shown to be embarrassingly parallel; we expect this to remain the case as long as the number of index partitions is greater than the number of cores (as is the case here; the dataset contains ≈ 22600 partitions worth of data). Reduced scalability occurs at high core counts due to I/O contention.

(higher CPU cost is canceled by lower I/O cost). We also determine the index build time, similar to total index size, is influenced by binning choice and data value distribution.

3.6.2.1 Parallel Indexing Scalability

In view of the exponential growth of extreme-scale scientific data, parallel scalability in any indexing system is of the utmost importance, and so we evaluate PIQUE's performance in this area.

To this end, we measure the time for PIQUE to build both a WAH and 3HD-tree index over the VPIC "energy" variable using between 256 to 4096 cores (stepping by factors of 2); since the input data is kept fixed while the number of cores varies, this evaluates strong scaling. The tests all use precision-3 binning and partition size 2^{23} elements (32 MB). We also run FastQuery under the same configuration as a reference point (note: an error occurred for the 256-core FastQuery case that we were unable to rectify, so FastQuery only reports timings for 512 to 4096 cores). The timings, which include both total time and compute-only time, are given in Figure 3.3.

We see that the CPU portion of the indexing process scales very well; this is to be expected given

its embarrassingly-parallel nature, provided there are sufficient partitions relative to cores, as is the case here ($\approx$ 22600 partitions worth of data). Past that point, the partition size would need to be reduced to maintain good strong scalability.

When considering total indexing time, however, scalability at high core counts is degraded. We surmise this is primarily due to limited I/O scaling, both intrinsic to the parallel filesystem and due to our current I/O model. To the former point, we used 96 OSTs through all of these tests (the maximum available), despite the growing number of concurrent writers. Thus, increased write contention, as well as run-to-run I/O variability, contribute to a relative increase in I/O time at higher core counts. In support of this, we note that the longest I/O time across any core in the 1024-core 3HD-tree run is $\approx 32.15s$, whereas it is $\approx 102.76s$ for 2048 and $\approx 76.05s$ for 4096 cores (a similar trend holds for WAH). Our current I/O model may also be a limiting factor, however: we use a single, shared file for the index written using MPI independent file I/O. While this is not an uncommon approach, the use of more advanced I/O middleware [27, 28, 54] and/or index aggregation [48] could help to reduce I/O contention and jitter even given a limited number of OSTs available.

We also note that FastQuery timings are higher than those for PIQUE. Since PIQUE's WAH implementation simply invokes the FastBit library, we believe this timing disparity must come from FastQuery's I/O handling, which is based on fully-collective HDF5 I/O. In contrast, PIQUE uses MPI independent I/O with very little synchronization (Section 3.5.3).

As a final note, MPI communication (other than MPI-IO) is negligible in all PIQUE tests; the most time spent by any core within a run ranges from 0.005 to 0.02 seconds, with no apparent upward trend at higher core counts. This includes not only send times, but also receive times, which includes any blocking time for indexing cores waiting for responses from the controller core. Thus, we infer that the dedicated controller approach is perfectly acceptable at these levels of concurrency, and is likely to continue to scale for some time.

3.7 Conclusion

Motivated by recent developments in scientific indexing that hint at the possibility of generalizing bitmap indexing, we pursue and develop a formal model encompassing the bitmap index and related indexing methods. We build a proof-of-concept implementation of this model, PIQUE, which supports a wide range of quantizations, RSet representations, and encodings from throughout the literature. In this vein, we also incorporate the hyperdyadic tree index described in Chapter 2 into this platform and do comparative evaluation relative to both the WAH-compressed bitmap as implemented within PIQUE, as well as versus an existing standalone parallel WAH indexing software, FastQuery. Our analysis shows promising performance: both for HD-trees, with index build times

comparable to the WAH index despite achieving substantially higher index compression; and for PIQUE, demonstrating strong scalability of the indexing platform as a whole, a benefit conferred upon all types of index supported.

In addition to the methodology and results presented here, another key contribution of our abstract indexing model and concrete PIQUE system is the range of possibilities they open. Avenues for future work are discussed further at the end of this work, in Section 6.2.

Chapter 4

Theoretical Modeling of HD-trees

4.1 Introduction

In Chapter 2, we develop the core algorithms for indexing and query processing over HD-trees, and demonstrate promising real-world performance. A next logical step is then to analyze the properties of HD-trees from a theoretical perspective, to gain further insight into which scenarios they are most suited to.

In particular, it should be possible to evaluate expected HD-tree compression ratios using various theoretical data models. This has been done previously for WAH-compressed bitmaps by generating random RSets using two random models: a Bernoulli process, and first-order Markov process [93]. Under the Bernoulli process, the membership of each RID in the random RSet is determined by independent weighted coin flips. Under the Markov process, the probability of a given RID being in the RSet is dependent on whether the previous RID was in the RSet, yielding some degree of RID clustering.

In this chapter, we apply these models to the HD-tree RSet representation, solving for expected RSet size. Building on these results, we further compute expected index sizes under various encodings and value distributions. Finally, we compare HD-trees and WAH using the theoretical model, giving insight into the relative strengths of each RSet representation.

The original formulation of these two RSet models is due to Wu et al. [93]. We contribute the following: an expanded explanation and refinement of the two models, restatement of the models for the generalized value-sliced index case, and derivation of expected index size for HD-trees under these models.

4.2 Preliminary Concepts

This chapter assumes knowledge of basic stochastic processes (especially Markovian processes) and related probability theory; an excellent introductory text covering these subjects has been written by Kleinrock [46].

Formally, both models we consider (Bernoulli and Markov) are *discrete-time stochastic processes* with binary state space $\{0,1\}$. That is, each model describes an infinite time series of *binary-valued random variables* $X(0), X(1), \ldots$, with $X(i) \in \{0,1\}$ representing the binary state at time i. Based on such a time series, we can define a random RSet R with an RID space of size n as $i \in R$ iff $X(i) = 1$ for all RIDs $i \in \mathbb{Z}_n$. It is then possible to compute the expected size of such a random RSet variable under a specific RSet representation, such as WAH compressed bitmap or HD-tree. Finally, a collection of random RSet variables can be used to represent the encoded RSets of an index, yielding the overall expected size.

Note: the terms "time" and "time series" are technical terms related to random processes, and bear no relation to the "time steps" of a simulation or dataset. Rather, "time" here is merely the positional index in the series of binary values produced by a process, and corresponds to RID.

Since many RSet representations are sensitive to "clustering" of RIDs (i.e., runs of consecutive RIDs in an RSet), a key property of a random process used to generate RSets is the *expected run length*. A *run* in a time series is defined as a *maximal* sequence of consecutive random variables that all have value "1"; the count of such variables is the run's *length* ℓ. Since it is maximal, it is followed by a "0" and not preceded by a "1" (i.e., preceded by "0" or starts at time 0).

For example, the time series

$$\begin{array}{lcccccccccccccc} i & : & 0, & 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & 12, & \ldots \\ X(i) & : & 0, & 0, & 0, & 1, & 1, & 0, & 0, & 1, & 1, & 1, & 0, & 1, & 0, & \ldots \end{array} \tag{4.1}$$

contains three runs: $i = 3,4$ (length 2); $i = 7,8,9$ (length 3); and $i = 11$ (length 1). The *expected run length at position i* of a process P is then the expected length of a run appearing at position i in the time series, that is,

$$\begin{aligned} &\mathrm{E}\big[\text{run length in } P \text{ at position } i\big] \\ &= \mathrm{E}[\ell \text{ s.t. } X(i+1), \ldots, X(i+\ell-1) = 1, X(i+\ell) = 0 \mid X(i) = 1, X(i-1) \neq 1]. \end{aligned} \tag{4.2}$$

For processes in steady state, the expected run length is position-invariant, and so it need not be predicated on a position i. Both models we consider (Bernoulli and Markov) will be shown to be

always in steady state, so in these cases we can simplify the expected run length as in Equation 4.3.

$$\begin{aligned}
&\mathrm{E}\big[\text{run length in } P \text{ at position } i\big] \\
&= \mathrm{E}[\ell \text{ s.t. } X(i+1),\ldots,X(i+\ell-1)=1, X(i+\ell)=0 \mid X(i)=1, X(i-1)\neq 1] \\
&= \mathrm{E}[\ell \text{ s.t. } X(1),\ldots,X(\ell-1)=1, X(\ell)=0 \mid X(0)=1].
\end{aligned} \tag{4.3}$$

4.2.1 Review of Relevant Index Encodings

The detailed workings of different index encodings (especially equality, range, interval, and binary component) are of central importance in this chapter, especially in Section 4.4. A comprehensive discussion of these encodings was already given in Section 3.4; instead of replicating it here, the reader is referred back to that earlier section to review the topic as needed.

4.3 Model Derivation

4.3.1 Bernoulli Process RSet Model

We first consider the case where an RSet is generated by a Bernoulli process (i.e., weighted coin flipping). The Bernoulli process takes one parameter, density $\delta \in [0,1]$, and generates a time series of *independent and identically distributed* binary random variables $X_\mathrm{b}(i)$ such that $X_\mathrm{b}(i)=1$ with probability δ and $X_\mathrm{b}(i)=0$ otherwise.

We define random RSet variable $R_\mathrm{b}(n,\delta)$ as an RSet over n possible RIDs constructed using the Bernoulli process with parameter δ such that

$$\Pr[i \in R_\mathrm{b}(n,\delta)] = \delta. \qquad (\text{for } i \in \mathbb{Z}_n) \tag{4.4}$$

The parameter δ is analogous to the *bit density* parameter d used in [93]; however, here we define it in terms of an abstract underlying random process, rather than a bitmap specific to the bitmap RSet representation.

4.3.1.1 Expected Run Length in the Bernoulli Process

Because as all random variables in a Bernoulli process are *independent and identically distributed*, the process is trivially always in steady state. For this reason, expected run length is position invariant,

and can be computed using Equation 4.3:

$$
\begin{aligned}
&\mathrm{E}\big[\text{run length in } P_{\mathrm{b}}(\delta)\big] \\
&= \mathrm{E}[\ell \text{ s.t. } X_{\mathrm{b}}(1),\ldots,X_{\mathrm{b}}(\ell-1)=1, X_{\mathrm{b}}(\ell)=0 \mid X_{\mathrm{b}}(0)=1] && \text{(Equation 4.3)} \\
&= \mathrm{E}[\ell \text{ s.t. } X_{\mathrm{b}}(1),\ldots,X_{\mathrm{b}}(\ell-1)=1, X_{\mathrm{b}}(\ell)=0] && \text{(all } X_{\mathrm{b}}(i) \text{ are i.i.d.)} \\
&= \sum_{\ell=1}^{\infty} \ell \Pr[X_{\mathrm{b}}(1),\ldots,X_{\mathrm{b}}(\ell-1)=1, X_{\mathrm{b}}(\ell)=0] \\
&= \sum_{\ell=1}^{\infty} \ell \delta^{\ell-1}(1-\delta) && \text{(all } X_{\mathrm{b}}(i) \text{ are i.i.d.)} \qquad (4.5) \\
&= \frac{1-\delta}{\delta} \sum_{\ell=1}^{\infty} \ell \delta^{\ell} \\
&= \frac{1-\delta}{\delta} \frac{\delta}{(1-\delta)^2} \\
&= \frac{1}{1-\delta}
\end{aligned}
$$

We have derived Equation 4.5 because it will be useful later on. At this point, we could also derive expected RSet size for the Bernoulli process. However, the Bernoulli process is just a special case of the Markov process (with $\sigma = 1$, as will be shown), so we forgo this derivation for brevity.

4.3.2 Markov Process RSet Model

The Bernoulli process model is a good first approximation of RSets generated by indexing. However, due to the spatio-temporal nature of many scientific datasets, data often exhibit some degree of *clustering* (defined as the tendency for nearby elements (in RID space) to have similar values). Clustering is preserved, and amplified, when (non-identity) quantization is used. A Bernoulli model cannot capture this effect, so we explore the use of a Markov process to model clustering.

We therefore consider using a two-state, discrete-time Markov chain, shown in Figure 4.1, as the underlying process for generating a random RSet. At each time step, a "1" state indicates that the corresponding RID is in the random RSet, whereas a state of "0" indicates it is not. By using a Markov process, rather than a memoryless Bernoulli process, we can model "clustering" of RIDs in the random RSet, and by extension, the spatial clustering of similar values in data variable.

We parameterize this Markov process with two parameters δ and σ (described below), and denote it as $P_{\mathrm{m}}(\delta,\sigma)$. This produces a time series $X_{\mathrm{m}}(0), X_{\mathrm{m}}(1),\ldots$, and a random RSet variable $R_{\mathrm{m}}(n,\delta,\sigma)$ may be constructed based on this series as $i \in R_{\mathrm{m}}(n,\delta,\sigma)$ iff $X_{\mathrm{m}}(i) = 1$ in process $P_{\mathrm{m}}(\delta,\sigma)$ for RIDs $i \in \mathbb{Z}_n$.

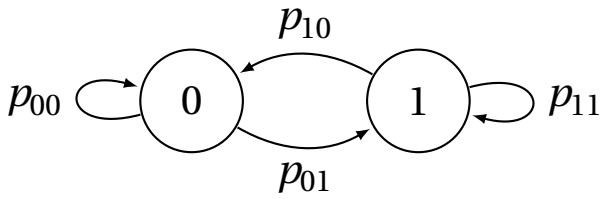


Figure 4.1 A Markov chain, capable of modeling simple RID clustering in an RSet. Transition probabilities p_{ij} are not intuitively meaningful, and so we compute them from two other, more intuitive parameters: *density* (δ) and *streakiness factor* (σ). These are described in Section 4.3.2.1.

To simplify the following analysis, we require that all transition probabilities be non-zero, guaranteeing ergodicity and the existence of a unique steady state. We also require that the initial distribution (probability of starting in each state) is equal to the steady state distribution, guaranteeing that the chain always remains in steady state. We denote the steady-state distribution vector as $\pi = \begin{bmatrix} \pi_0 & \pi_1 \end{bmatrix}$.

4.3.2.1 Parameterization

The Markov model may be fully defined by two parameters. However, the intrinsic parameters (p_{00}, p_{01}, etc.) are not easily interpretable. Instead, we select two meaningful parameters, *density* δ and *streakiness factor* σ, then derive the transition probabilities.

Just as with the Bernoulli process, we define density $\delta \in (0,1)$ as the probability of any given time step being in state "1." This probability is constant at any position i in the time series due to the always-steady-state property. Thus, we can say

$$\delta = \Pr[X_{\mathrm{m}}(i) = 1]. \qquad \text{(for any } i\text{)} \tag{4.6}$$

The second parameter, *streakiness factor* $\sigma \in [1, \infty)$, is defined as the ratio of expected run length in $P_{\mathrm{m}}(\delta, \sigma)$ to the expected run length in the analogous Bernoulli process $P_{\mathrm{b}}(\delta)$. That is, σ indicates *how many times longer* a run in the Markov process is expected to be versus in a Bernoulli process with the same density. In the associated random RSet $P_{\mathrm{m}}(\delta, \sigma)$, this translates to a scaling of runs of consecutive RIDs[1].

As with the Bernoulli process, expected run length in the Markov process is position-invariant. This is due to the *Markov property*: $X_{\mathrm{m}}(i+1)$ is only dependent on $X_{\mathrm{m}}(i)$, which means the conditional in the expected run length ($\mathrm{E}[\ldots \mid X_{\mathrm{m}}(i) = 1]$) makes the whole expected value position-invariant.

Thus, σ is a constant for the Markov process, and can be stated as:

$$\sigma = \frac{\mathrm{E}\big[\text{run length in } P_{\mathrm{m}}(\delta,\sigma)\big]}{\mathrm{E}\big[\text{run length in } P_{\mathrm{b}}(\delta)\big]}. \tag{4.7}$$

4.3.2.2 Comparison with Parameters from Wu et al.

While the density parameter δ used here is a direct analogue of *bit density d* in Wu et al. [93] (though generalized from bitmaps to RSet representations), our definition of streakiness factor σ differs from the previous use of *clustering factor f*. In prior work, f is defined as the expected run length, i.e., $f = \mathrm{E}\big[\text{run length in } P_{\mathrm{m}}(\delta,\sigma)\big]$. In contrast, σ is the *ratio* of the expected run length under the Markov model *to the expected run length under the Bernoulli model*. Put simply, f determines how long the average run is, whereas σ determines how *unusually long* the average run is (versus the no-clustering case).

While both definitions yield equally-valid results, we find σ more useful because it is more "orthogonal" to δ than f is. In particular:

1. $\sigma = 1$ always reduces to the Bernoulli model (whereas the value of f with this property depends on δ), and

2. if we fix σ and vary δ, the random RSets generated will have a similar degree of clustering (whereas under fixed f, smaller δ values exhibit progressively greater degrees of clustering).

4.3.2.3 Expected Run Length in the Markov Process

We will need the expected run length of the Markov process below, so we take a moment to derive it in Equation 4.8. This equation is formulated in terms of the transition probabilities from Figure 4.1.

[1] Because translation from $P_{\mathrm{m}}(\delta,\sigma)$ to $R_{\mathrm{m}}(n,\delta,\sigma)$ truncates the process at n, the true expected run length of RIDs in the RSet will be (very slightly) less than that of the (infinite) underlying process; however, for typical n, the difference is negligible, and σ still has the intended effect of modulating clustering.

Recall that the Markov process is position-invariant, so we can utilize Equation 4.3.

$$\begin{aligned}
&\mathrm{E}\big[\text{run length in } P_{\mathrm{m}}(\delta,\sigma)\big] \\
=&\mathrm{E}[\ell \text{ s.t. } X_{\mathrm{b}}(1),\ldots,X_{\mathrm{b}}(\ell-1)=1, X_{\mathrm{b}}(\ell)=0 \mid X_{\mathrm{b}}(0)=1] && \text{(Equation 4.3)} \\
=&\sum_{\ell=1}^{\infty} \ell \cdot \Pr[X_{\mathrm{m}}(1),\ldots,X_{\mathrm{m}}(\ell-1)=1, X_{\mathrm{m}}(\ell)=0 \mid X_{\mathrm{m}}(0)=1] \\
=&\sum_{\ell=1}^{\infty} \ell \cdot p_{11}^{\ell-1} \cdot p_{10} && \text{(Markov property)} \\
=&\frac{p_{10}}{p_{11}} \sum_{\ell=1}^{\infty} \ell \cdot p_{11}^{\ell} \\
=&\frac{p_{10}}{p_{11}} \frac{p_{11}}{(1-p_{11})^2} \\
=&\frac{p_{10}}{p_{11}} \frac{p_{11}}{p_{10}^2} \\
=&\frac{1}{p_{10}}
\end{aligned} \tag{4.8}$$

4.3.2.4 Deriving Markov Chain Transition Probabilities

We now derive the transition probabilities for the Markov chain of Figure 4.1 from parameters δ and σ. First, we develop expressions for the steady-state probabilities π_0, π_1:

$$\begin{bmatrix}\pi_0 & \pi_1\end{bmatrix}\begin{bmatrix}p_{00} & p_{01} \\ p_{10} & p_{11}\end{bmatrix} = \begin{bmatrix}\pi_0 & \pi_1\end{bmatrix}. \tag{4.9}$$

Solving for π_0 and π_1:

$$\begin{aligned}
\pi_0 &= \frac{p_{10}}{p_{01}+p_{10}} \\
\pi_1 &= \frac{p_{01}}{p_{01}+p_{10}}.
\end{aligned} \tag{4.10}$$

Since the Markov chain is always in steady state, we can relate δ to the transition probabilities:

$$\begin{aligned}
\delta &= \Pr[X_{\mathrm{m}}(i)=1] && \text{(Equation 4.6)} \\
&= \pi_1 && \text{(always steady state)} \\
&= \frac{p_{01}}{p_{01}+p_{10}}. && \text{(Equation 4.10)}
\end{aligned} \tag{4.11}$$

Next, we show $\sigma = \frac{1}{p_{01}+p_{10}}$ by expanding Equation 4.7:

$$\begin{aligned}
\sigma &= \frac{\mathrm{E}\big[\text{run length in } P_{\mathrm{m}}(\delta,\sigma)\big]}{\mathrm{E}\big[\text{run length in } P_{\mathrm{b}}(\delta)\big]} && \text{(Equation 4.7)} \\
&= \frac{\frac{1}{p_{10}}}{\frac{1}{1-\delta}} && \text{(Equations 4.5 and 4.8)} \\
&= \frac{1-\delta}{p_{10}} \\
&= \frac{1-\frac{p_{01}}{p_{01}+p_{10}}}{p_{10}} && \text{(Equation 4.11)} \\
&= \frac{\frac{p_{10}}{p_{01}+p_{10}}}{p_{10}} \\
&= \frac{1}{p_{01}+p_{10}}
\end{aligned} \tag{4.12}$$

It is now possible to derive any transition probability from Equations 4.12 and 4.11:

$$\begin{aligned}
p_{01} &= \frac{\delta}{\sigma} \\
p_{00} &= 1-\frac{\delta}{\sigma} \\
p_{10} &= \frac{1-\delta}{\sigma} \\
p_{11} &= 1-\frac{1-\delta}{\sigma}
\end{aligned} \tag{4.13}$$

4.3.2.5 Formalization of the HD-tree Structure

Before finally computing expected kHD-tree size, we take one last step and formalize the structure of the HD-tree.

In what follows, we use $c = 2^k$, as this quantity appears often. Also note the use of notation $[a \,..\, b]$ for an *integer interval* denoting the set $\{a, a+1, \ldots, b\}$; variants for open (e.g., $(a \,..\, b)$) and half-open (e.g., $[a \,..\, b)$ or $(a \,..\, b]$) are defined in the expected way.

For a given RID space $\mathbb{Z}_n$, let $\ell = \lceil \log_k n \rceil$. We then define the set of *characterizing (hyperdyadic) intervals* $C_k(n)$ for this RID space at a given k as

$$C_k(n) = \left\{ I_{i,j} = \big[j c^i \,..\, (j+1)c^i \big) \mid i \in [1 \,..\, \ell],\, j \in \left[0 \,..\, \left\lceil \frac{n}{c^i} \right\rceil \right) \right\}. \tag{4.14}$$

It can be seen that $C_k(n)$ is the set of all hyperdyadic intervals that *intersect* but do not *contain*

the RID space, with the exception that $I_{\ell,0} \in C_k(n)$; $I_{\ell,0}$ does contain the RID space ($I_{\ell,0}$, and is the smallest hyperdyadic interval to do so. That is, we can say

$$C_k(n) = \{I_{\ell,0}\} \cup \{I_{i,j} \mid I_{i,j} \cap \mathbb{Z}_n \neq \emptyset \wedge I_{i,j} \cap \mathbb{Z}_n \neq \mathbb{Z}_n\}. \tag{4.15}$$

(Note: hereafter in this section, unless otherwise stated, the term "interval" is assumed to mean "characterizing hyperdyadic interval.")

Every interval $I_{i,j}$ has some *child* interval(s) $I_{i-1,cj}, \ldots, I_{i-1,cj+h}$ for some $h \in [1\,..\,c]$ (except when $i = 1$, in which case no children exist). Similarly, every interval $I_{i,j}$, except $I_{\ell,0}$, has a parent interval $I_{i+1,\left\lfloor \frac{j}{c} \right\rfloor}$. In keeping with tree terminology, we term $I_{\ell,0}$ the *root interval* (since it has no parent) and intervals $I_{1,j}$ *leaf intervals* (since they have no children).

We say that an interval is *empty* or *filled* with respect to an RSet iff that RSet contains *no* or *all* RIDs in that interval, respectively. For our purposes, we only consider those RIDs within the RID space of the RSet, so more specifically, empty/filled means no/all RIDs in the *intersection of the interval and the RID space* are present in the RSet. More formally, with respect to a region R over an RID space of size n,

$$\begin{aligned} \text{Hyperdyadic interval } I_{i,j} = \left[j\,c^i\,..\,(j+1)c^i\right) \text{ is } \textit{empty} \text{ in RSet } R \text{ iff } \forall\, r \in I_{i,j} \cap [0\,..\,n)\ (r \notin R) \\ \cdots \text{ is } \textit{filled} \text{ in RSet } R \text{ iff } \forall\, r \in I_{i,j} \cap [0\,..\,n)\ (r \in R). \end{aligned}$$

We say an interval $I_{i,j}$ is *pure* with respect to some RSet if it is either empty or filled in that RSet, *and does not contain the entire RID space* (i.e., $[0\,..\,n) \not\subseteq I_{i,j}$; thus, the root interval is never considered pure). We say a pure interval is *maximal* if its parent is not pure. It is easy to see that all children (and descendants) of a pure interval are also pure, and by contrapositive, no parents (or ancestors) of a non-pure interval are pure (similarly for empty and filled intervals). Thus, on any path from a pure interval to the (non-pure) root interval, there is exactly one transition from pure to non-pure. It then follows that every pure interval has a unique maximal pure ancestor interval.

Using this terminology, we can restate the structure a kHD-tree for an RSet with RID space $\mathbb{Z}_n$: a kHD-tree is a sequence of *words* corresponding to characterizing intervals such that the sequence represents the *breadth-first traversal of all non-pure intervals* in the associated region (considering the characterizing intervals as a tree according to parent/child relationships). This is the case because each word describes an interval, has a child word for each child interval that is not pure, and child words are inserted into the HD-tree in FIFO (i.e., queue) order, giving rise to the breadth-first order. This leads to three corollary properties:

1. every *non-pure* interval is represented *explicitly* by a unique HD-tree word,

2. every *maximal pure* interval is represented *implicitly* by a 0 or 1 code in its parent (non-pure) interval's word (and does thus not require its own, separate word), and

3. every *non-maximal pure* interval is represented *implicitly* by its maximal pure ancestor (and thus does not require its own, separate word).

Finally, note that HD-tree words at tree level i (counted from the bottom starting at 1) correspond to intervals of the form $I_{i,j}$, and words at a given level (i.e., siblings) are ordered from least to greatest j.

Note that there are two corner cases to these properties, which are successfully resolved by the careful definition of "pure" intervals.

The first is the root HD-tree word, which is always present even if the entire RSet is empty or filled. This is covered by the non-containment condition in the definition of "pure," which causes the root interval to always be considered non-pure; thus, the root interval correctly falls under case 1 in the above list.

The second case is when a word's corresponding interval is "truncated" by n, that is, the interval extends beyond n. Such an interval is considered pure iff the portion of the interval within the RID space is empty or filled, implying no corresponding word in the HD-tree due to case 2 or 3 above. However, it's possible an HD-tree construction algorithm would nevertheless allocate a word to such a truncated interval, filling the in-bounds part with "1" codes and the out-of-bounds portion with "0" codes (possibly with a "2" code at the boundary). This case can be avoided, however, by a process we term "code smearing": choosing arbitrary codes for out-of-bounds portions of the HD-tree so as to eliminate these spurious mixed words. That this may cause the HD-tree to represent some RIDs greater than or equal to n is not an issue, since n is known and such out-of-bounds RIDs may easily be ignored.

A small example of code smearing: consider a 2HD-tree over RSet $\{0, 1, \ldots, 13\}$ with RID space $n = 14$. The characterizing interval $I_{1,3} = [12 \,..\, 16)$ is considered pure (filled), since in-bounds RIDs $\{12, 13\}$ are all in the region; thus, it should not have a corresponding word. Yet, a valid representing 2HD-tree is "1112 1100," with the second word corresponding to $I_{1,3}$, contradicting this inference. However, by applying "code smearing," we can construct another valid 2HD-tree: "1112 1111." In this case, we fill the out-of-bounds portion of the last word with "1" codes, even though RIDs 14, 15 are not actually present in the region; this is fine since these codes are implicitly ignored as out-of-bounds. This HD-tree is then simplified to "1111," and now $I_{1,3}$ is represented implicitly without its own HD-tree word, as expected.

In practice, code smearing need not be used; the space savings is minimal. However, its use simplifies analysis, as now our definition of "non-pure" intervals corresponds exactly to those words that are present in the corresponding HD-tree, so its use is assumed in this chapter.

4.3.2.6 Computing Expected HD-tree Size

Now that the Markov model is fully characterized, and we have a more rigorous description of the HD-tree, we can compute expected kHD-tree size.

Consider the random RSet $R_{\mathrm{m}}(n,\delta,\sigma)$ generated by Markov process $P_{\mathrm{m}}(\delta,\sigma)$. For any characterizing interval $I_{i,j} = \left[j\,c^i \,..\, (j+1)c^i\right)$, let $W_{i,j}$ be an indicator variable (i.e., random binary variable) such that $W_{i,j} = 0$ iff interval $I_{i,j}$ is pure in $R_{\mathrm{m}}(n,\delta,\sigma)$, and $W_{i,j} = 1$ otherwise.

Based on Section 4.3.2.5, for each characterizing interval $I_{i,j}$, there exists a unique corresponding HD-tree word iff that interval is non-pure, or equivalently, iff $W_{i,j} = 1$. Thus, we can say that the expected number of HD-tree words used represent $R_{\mathrm{m}}(n,\delta,\sigma)$ is

$$\begin{aligned} \mathrm{E}[k\text{HD-tree words}] &= \sum_{I_{i,j}\in C_k(n)} W_{i,j} \\ &= \mathrm{E}\left[\sum_{i=1}^{\ell}\sum_{j=0}^{\left\lceil \frac{n}{c^i}\right\rceil -1} W_{i,j}\right] \qquad \text{(Equation 4.14)} \end{aligned} \tag{4.16}$$

Assuming the use of dense suffix coding (see Section 2.3.1, each word consists of $2\cdot c$ bits, except words in the lowest tree level (level 1), which require c bits each. Denote this bits-per-word cost at level i as

$$B_i = \begin{cases} c & \text{if } i = 1 \\ 2c & \text{otherwise} \end{cases} \tag{4.17}$$

Combining all of this, we obtain a first expression for expected kHD-tree size of

$$\begin{aligned} \mathrm{E}[k\text{HD-tree}] &= \mathrm{E}\left[\sum_{i=1}^{\ell}\sum_{j=1}^{\left\lceil \frac{n}{c^i}\right\rceil} W_{i,j}B_i\right] \\ &= \sum_{i=1}^{\ell} B_i \sum_{j=1}^{\left\lceil \frac{n}{c^i}\right\rceil} \Pr\left[W_{i,j}=1\right]. \end{aligned} \tag{4.18}$$

This expression is exact, and is used to construct the plots in Section 4.5. However, if expanded as-is, the expression becomes very complicated due to the corner cases of the root word and truncated words, discussed previously. Such an expression would be unhelpful for deriving intuition about the HD-tree. Therefore, we make the two following approximations to simplify the equation.

The first exception is $W_{\ell,0}$, the root word, which is always present even if its interval is empty/filled. Thus, $\Pr\left[W_{\ell,0}=1\right]=1$. Instead of maintaining this special case in the expected value expression, we

instead model the root word using the same probability as a "normal" word. This approximation is almost exact, underestimating only the unlikely case that the entire RSet is empty or filled, and that by only 1 word $= 2c$ bits.

The second exception is truncated words at the end of a level i, as discussed before. Recall that we consider a word's interval filled (empty) iff all RIDs *less than n* are present (absent). Because of this, truncated words have a greater than normal probability of being pure, as fewer RIDs need be present/absent. This edge case also complicates the expression, so we approximate by also considering these words as "normal," non-truncated words. Though this represents a slight overestimate in expected size, it is strictly less than $\log_c(n)$ words $= 2c\log_c(n) - c$ bits too large in the worst case. This error is negligible unless δ is extremely close to 0 or 1, on the order of $\frac{1}{n}$ or $1-\frac{1}{n}$; as we will show, realistic indexing cases induce $\delta \gg \frac{1}{n}$, so this is generally not an issue.

Now, we compute the probability of a given characterizing interval being pure in an HD-tree under these approximations. This may be computed from the Markov chain steady state and transition probabilities as

$$\begin{aligned}
\Pr\left[W_{i,j}=1\right] &\approx \Pr\left[I_{i,j} \text{ pure in } R_{\mathrm{m}}(n,\delta,\sigma) \,\middle|\, I_{i,j} \text{not root/truncated}\right] \\
&= 1-\Pr\left[I_{i,j} \text{ pure } \middle| \ldots\right] \\
&= 1-\Pr\left[I_{i,j} \text{ empty } \middle| \ldots\right]-\Pr\left[I_{i,j} \text{ filled } \middle| \ldots\right] \\
&= 1-\left(\pi_0 p_{00}^{c^i-1}\right)-\left(\pi_1 p_{11}^{c^i-1}\right) \\
&= 1-(1-\delta)\left(1-\frac{\delta}{\sigma}\right)^{c^i-1}-\delta\left(1-\frac{1-\delta}{\sigma}\right)^{c^i-1}. \qquad \text{(Equations 4.11, 4.13)}
\end{aligned} \tag{4.19}$$

If we consider an k-octree using dense suffix coding each word consists of $2\cdot c$ bits, except words in the lowest level, which take c bits each. Denote this bits-per-word cost as $B_i = (c \text{ if i} = 1, 2c \text{ otherwise})$ for level i. Combining word probabilities and bits-per-word sizes, the expected size in bits of the

k-HD-tree for $R_{\mathrm{m}}(n,\delta,\sigma)$ can be computed as in Equation 4.18.

$$\begin{aligned}
\mathrm{E}[k\text{HD-tree size for } R_{\mathrm{m}}(n,\delta,\sigma)] &= \mathrm{E}\left[\sum_{i=1}^{\ell}\sum_{j=1}^{\left\lceil\frac{n}{c^i}\right\rceil} W_{i,j}B_i\right] \\
&= \sum_{i=1}^{\ell} B_i \sum_{j=1}^{\left\lceil\frac{n}{c^i}\right\rceil} \Pr\left[W_{i,j}=1\right] \\
&\approx \sum_{i=1}^{\ell} B_i \sum_{j=1}^{\left\lceil\frac{n}{c^i}\right\rceil} 1-(1-\delta)\left(1-\frac{\delta}{\sigma}\right)^{c^i-1} - \delta\left(1-\frac{1-\delta}{\sigma}\right)^{c^i-1} \\
&\approx \sum_{i=1}^{\ell} B_i \frac{n}{c^i}\left(1-(1-\delta)\left(1-\frac{\delta}{\sigma}\right)^{c^i-1} - \delta\left(1-\frac{1-\delta}{\sigma}\right)^{c^i-1}\right)
\end{aligned} \tag{4.20}$$

The final step of Equation 4.18 makes one last approximation, namely, replacing $\left\lceil\frac{n}{c^i}\right\rceil$ with $\frac{n}{c^i}$. This amounts to underestimating by at most one HD-tree word per level (with order $\log n$ levels). Actually, this approximation is, in a sense, the reverse of the overestimation of treating truncated words as whole words, as it effectively treats the last word of each layer as potentially shorter than a full word.

4.3.3 Comparison between Expected Sizes for WAH and kHD-tree

Restating the approximate expected WAH-compressed bitmap size from Wu et al. [93] (adapted from Equation 3 in that paper) using our parameter definitions, we have:

$$\mathrm{E}[\text{WAH size in bits}] = w\frac{n}{w-1}\left(1-(1-\delta)\left(1-\frac{\delta}{\sigma}\right)^{2(w-1)-1} - \delta\left(1-\frac{1-\delta}{\sigma}\right)^{2(w-1)-1}\right). \tag{4.21}$$

where w is the word size (typically 32 or 64).

A particular expression appears prominently in both Equation 4.20 and 4.21:

$$x\frac{n}{y}\left(1-(1-\delta)\left(1-\frac{\delta}{\sigma}\right)^{z-1} - \delta\left(1-\frac{1-\delta}{\sigma}\right)^{z-1}\right) \tag{4.22}$$

(for some value of x, y, and z). Based on our earlier derivation for HD-trees, we can say that the main body of this expression (everything after $x\frac{n}{y}$) is the probability of a (not necessarily characterizing) interval of length z being non-pure (the exponent $z-1$ appears because $z-1$ state transitions of "0"-to-"0" or "1"-to-"1" are required in the Markov chain to construct a pure interval

of length z). Then, n/y is the number of such intervals considered by the data structure (WAH or HD-tree), and x is the size in bits of the "word" needed to represent an interval if it is non-pure.

Recall that WAH is run-length compression, which divides a sequence of bits into mutual disjoint and exhaustive "counting groups" of size $w-1$, and stores them using two types of words of size w bits:

1. "literal words," each of which consists of a "fill flag" bit of 0 followed by the $w-1$ bits of the counting group, and

2. "fill words," each of which consists of a "fill flag" bit of 1, a "fill bit" indicating whether the interval represented is empty (0) or filled (1), and a "count" field of $w-2$ bits representing an integer indicating the number of consecutive counting groups that are pure.

For WAH, the intervals of interest of are length $z = 2(w-1)$, or two "counting groups" in length. Whenever two consecutive counting groups are pure, the second counting group will be implicitly represented as part of a run encoded by either the first counting group's word or some earlier counting group's word. Otherwise, the two consecutive counting groups are *not* pure, and the second will require a new word, either fill or literal depending on whether the second counting group and its successor are both pure. The number of such intervals considered is approximately $\frac{n}{y} = \frac{n}{w-1}$ (every pair of consecutive counting groups), and the size (in bits) of the word incurred by a non-pure pair of counting groups is $x = w$.

The structure of the HD-tree expected size expression is related. For HD-trees, the intervals of interest are the characterizing intervals of length $z = c^i$ for $i \in \{1, \ldots, \ell\}$. The number of such intervals are about $\frac{n}{y} = \frac{n}{c^i}$ for the same values of i, and the size in bits of each word is $z = 2c$, except for $i = 1$, where it is $z = c$.

Clearly, WAH with $w = 32$ (a typical value) requires longer runs ($z = 62$) to eliminate words and achieve compression, as compared to HD-trees with typical k values of 2, 3, or 4 ($z = 4$, 8, or 16 respectively at the lowest tree level). That is, we can say that WAH is coarser-grained compression. However, HD-trees have multiple, overlapping levels, as opposed to the more linear approach of WAH, leading to possible representation overhead (even though the maximum words per level falls off exponentially by a factor of $c = 2^k$ per level). Thus, there is not a clear winner between WAH and HD-tree sizes based on examination of these formulae alone.

Therefore, we explore these models further by plotting expected index sizes for WAH and HD-trees under various parameter combinations in Section 4.5. First, however, we show how expected RSet size determines index size under various bin distribution and index encoding assumptions.

4.4 Expected Total Index Sizes

In prior sections, we have established a formula for expected HD-tree size. A full HD-tree index, however, is composed of many encoded RSets, each of which is an HD-tree. Furthermore, the parameters for each of these RSets may be different, especially δ. In this section, we derive final expressions for expected index sizes.

First, it is important to note that, for a given random dataset, the sizes of encoded RSets in the corresponding index are not *statistically independent*. That is, the contents of any one encoded RSet influences probabilities for the contents of the other encoded RSets. For example, in an equality-encoded index, if a given RID exists in one encoded region (i.e., bin), it is guaranteed to *not* exist in any other encoded region (i.e., $r \in E_i \rightarrow \forall\, j \neq i\colon \Pr\big[r \in E_j\big] = 0$). Thus, deriving a closed-form probability distribution for a random dataset's index would be extremely difficult, if not impossible. However, due to the *linearity* property of expectation, we can compute the expected index size as the sum of the expected sizes of all encoded RSets, ignoring their statistical dependence.

In what follows, let the expected size of an index over n RIDs, with number of bins b, bin distribution B, RSet representation R, and index encoding E be denoted as $\mathrm{E}[\mathcal{I}(n,b,B,R,E)]$. We keep n, b, and R general, and explore different options for B and E. Specifically, we consider general bin distribution and uniform bin distribution, and we consider index encodings *equality*, *range*, *interval* [17], and *binary component* (that is, base-$(2,\ldots,2)$ component encoding) [16, 98].

For an equality-encoded index, each encoded RSet is simply a bin RSet, that is, the set of RIDs that fall into a certain bin. If b_i is the fraction of RIDs residing in the ith bin, and if σ is a constant intra-bin clustering factor, we have:

$$\begin{aligned} &\mathrm{E}\big[\mathcal{I}(n, b, b_i \text{ distribution with } \sigma \text{ clustering}, R, \text{equality})\big] \\ &= \sum_{i=1}^{b} \mathrm{E}[R \text{ size for } R_{\mathrm{m}}(n, b_i, \sigma)] \end{aligned} \tag{4.23}$$

For the specific case of uniform bin distribution ($b_i = \frac{1}{b}$), we have:

$$\begin{aligned} &\mathrm{E}\big[\mathcal{I}(b, \text{uniform with } \sigma \text{ clustering}, R, \text{equality})\big] \\ &= \sum_{i=1}^{b} \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \frac{1}{b}, \sigma\right)\right] \\ &= b\,\mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \frac{1}{b}, \sigma\right)\right] \end{aligned} \tag{4.24}$$

For a range-encoded index consists of $b-1$ encoded regions, where the ith encoded RSet

corresponds to the set of RIDs falling into any of the first i bins. Thus, we have:

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(n,b,b_i \text{ distribution with } \sigma \text{ clustering}, R, \text{range})\big]\\ &= \sum_{i=1}^{b-1} \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \sum_{j=1}^{i} b_j, \sigma\right)\right]\end{aligned} \tag{4.25}$$

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(b,\text{uniform with } \sigma \text{ clustering}, R, \text{range})\big]\\ &= \sum_{i=1}^{b-1} \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \frac{i}{b}, \sigma\right)\right]\end{aligned} \tag{4.26}$$

An interval-encoded index consists of $\lceil \frac{b}{2} \rceil$ encoded RSets, with the ith RSet each covering bins $\big[i \,..\, i + \lfloor \frac{b}{2} \rfloor\big)$. Therefore:

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(n,b,b_i \text{ distribution with } \sigma \text{ clustering}, R, \text{interval})\big]\\ &= \sum_{i=1}^{\lceil \frac{b}{2} \rceil} \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \sum_{j=i}^{i+\lfloor \frac{b}{2} \rfloor - 1} b_j, \sigma\right)\right]\end{aligned} \tag{4.27}$$

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(b,\text{uniform with } \sigma \text{ clustering}, R, \text{interval})\big]\\ &= \left\lceil \frac{b}{2} \right\rceil \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \frac{\lfloor \frac{b}{2} \rfloor - 1}{b}, \sigma\right)\right]\end{aligned} \tag{4.28}$$

Finally, we consider a binary component-encoded index. Let $L(n,i)$ denote the set of integers $j \in \{1,\ldots,n\}$ such that $(j-1)\,\&\,2^i = 0$, where & represents bitwise AND. Then, a binary component-encoded index has $\ell = \lceil \log_2 \rceil$ "levels" (encoded RSets), such that the ith level of covers bins $L(n,i)$. Thus,

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(n,b,b_i \text{ distribution with } \sigma \text{ clustering}, R, \text{binary component})\big]\\ &= \sum_{i=1}^{\ell} \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \sum_{j \in L(n,i)} b_j, \sigma\right)\right]\end{aligned} \tag{4.29}$$

$$\begin{aligned}&\mathrm{E}\big[\mathcal{I}(b,\text{uniform with } \sigma \text{ clustering}, R, \text{binary component})\big]\\ &= \ell\, \mathrm{E}\left[R \text{ size for } R_{\mathrm{m}}\left(n, \frac{1}{2}, \sigma\right)\right]\end{aligned} \tag{4.30}$$

All of these expressions tell us important values for δ span the interval $(0,1)$: $\delta = \frac{1}{b}$ for equality encoding, $\delta \approx \frac{1}{2}$ for interval and binary component encoding, and $\delta \in \left(\frac{1}{b}, \frac{b-1}{b}\right)$ for range encoding

under a uniform value distribution. Therefore, any RSet representation (in particular, HD-trees and WAH) should ideally achieve good compression across a large range of RSet densities. In the next section, we measure to what extent this is the case.

4.5 Results

In order to examine HD-trees more closely in comparison with WAH, we perform a series of experiments. In each, we vary δ and/or σ, and plot expected sizes for kHD-tree and WAH. Markov steady-state and transition probabilities, compressing each using kHD-tree and WAH, and reporting the mean sizes in bits actually observed. In all cases, we use a dataset size (number of elements) of $n = 2^{26}$, as this is a typical value for *index partition size* (see Section 3.5.3 for details on the use of index partitioning for parallelism and coping with large datasets).

4.5.1 Uniform Bin Distribution, Equality Encoding

Our first experiment measures index size under uniform bin distribution and equality encoding while varying the number of bins, as per Equation 4.24. This section assumes a fixed $\sigma = 1$, that is, uses the Bernoulli process to model (the lack of) clustering; Section 4.5.3 explores the effect of varying σ.

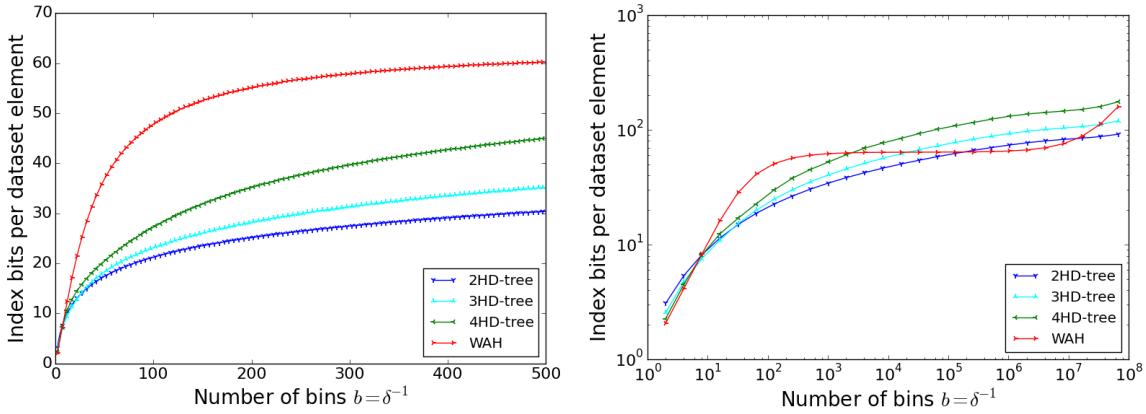
The horizontal axis plots bin count $b = \delta^{-1}$, with the Y axis reporting the number of index bits *per data element* (i.e., index size divided by n). We consider each plot in turn.

For Figure 4.2, we split the plot into two views with different X-axis (i.e., bin count) scales to give additional insight. Figure 4.2a uses a linear axis scale for bin counts, $b \in [2\,..\,500]$, whereas Figure 4.2b uses a log scale to extend the range of bin counts to $b = 2^i$ for $i \in [1\,..\,26]$.

Figure 4.2a gives theoretical evidence supporting a main conclusion of Section 2.4, that the HD-tree provides a substantial compression gain over WAH. Here we see similar compression between WAH and HD-trees at low bin count, after which HD-trees supersede WAH with a widening gap as bin count increases (or, equivalently, as per-RSet density decreases). The maximum compression gains of HD-tree over WAH in this plot are: $\approx 2.26x$ at 105 bins for 2HD-tree, $\approx 2.07x$ at 83 bins for 3HD-tree, and $\approx 1.82x$ at 60 bins for 4HD-tree.

Figure 4.2b, however, gives us a broader view and highlights several other behaviors. WAH outperforms kHD-trees when the bin count is very low (< 8), as before, but also when bin count is very high, as well (> 2048 or more, depending on k). These two cases merit closer examination.

Higher WAH compression at very low bin count (per bin density near 0.5) is more or less an indication of relative worst-case compression: the addition of the "bitmap" curve in the plot shows



(a) Linear-scale bin counts, $b = [2 \,..\, 500]$ **(b)** Log-scale bin counts, $b = \{2^i\}$ for $i \in [1 \,..\, 26]$

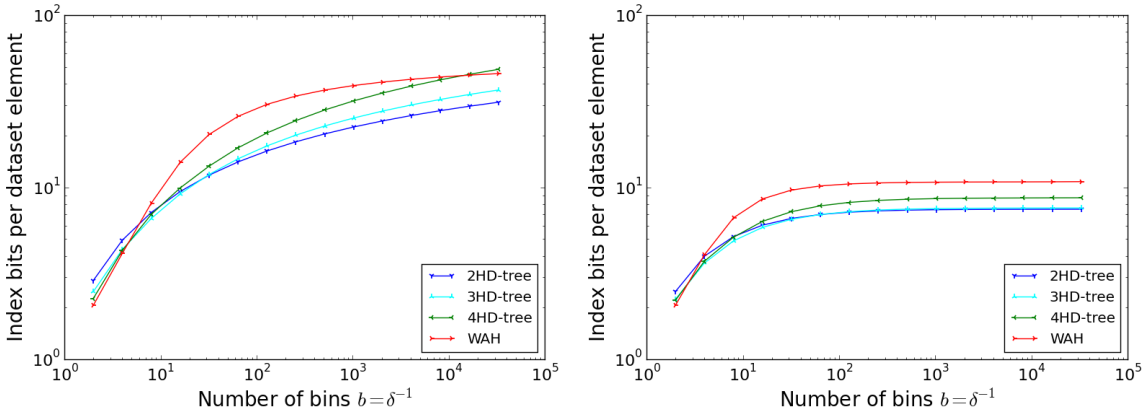
Figure 4.2 Comparison of expected index size under HD-tree and WAH RSet representations, for a dataset with a *uniform* bin distribution, *equality* encoding, and no streakiness ($\sigma = 1$)
. A fine-grained, narrow range of bin counts is given in Figure 4.2a, with a coarser but broader range of bin counts given in Figure 4.2b.

that all four methods are outperformed by a simple *uncompressed* bitmap RSet representation at these bin counts. This is because pure intervals of sufficient length for compression are very uncommon at densities so close to 0.5. WAH performs better because its worst case compression ratio is better at $\sim \frac{w-1}{w}$, whereas kHD-tree has a worst case of $\sim \frac{2^k-1}{2^k+1}$. For the values of $w = 32$ and $k = 2,3,4$ used, these amount to ~ 0.97 for WAH and ~ 0.6, ~ 0.78, and ~ 0.88 for HD-trees with $k = 2,3,4$, respectively. This phenomenon is arguably of little practical value, however, since likely no form of index compression is advisable in such a case.

Better WAH compression at very high bin counts is similarly explained by WAH having better compression in very sparse cases. When present RIDs are very far apart, WAH has a single initial fill word, and two words per RID (a literal word containing the RID, and a fill word spanning the gap to the next literal word), for a total of between 64 and 96 amortized bits per dataset element (assuming $w = 32$). In contrast, HD-trees allocate an entire tree path of words for each present RID, which costs $2^{k+1}\left\lceil \log_{(2^k)}(n) \right\rceil - 2^k \approx \frac{2^{k+1}}{k} \log_2(n) - 2^k$ bits per RID[2], which is 192 bits per RID for $k = 4$ and $n = 2^26$ as in Figure 4.2b, for instance. Some prefix of each path is shared with other paths, reducing the actual per-RID storage cost, but at such low density path sharing is minimal, and thus overall this results in higher overhead for HD-trees versus WAH.

From an information-theoretic perspective, these crossover points are unsurprising: every

[2] 2^{k+1} bits per word, $\left\lceil \log_{(2^k)}(n) \right\rceil$ tree levels, and the lowest-level word uses dense suffix coding, eliminating 2^k bits.



(a) Zipfian bin density distribution, parameter $s = 1.1 \rightarrow \delta_i \propto i^{-1.1}$

(b) Zipfian bin density distribution, parameter $s = 2 \rightarrow \delta_i \propto i^{-2}$

Figure 4.3 Comparison of expected index size under HD-tree and WAH, for a dataset with a *Zipfian* bin distribution, *equality* encoding, and no streakiness ($\sigma = 1$). Figures 4.3a and 4.3b show the relationship under different characterizing exponents.

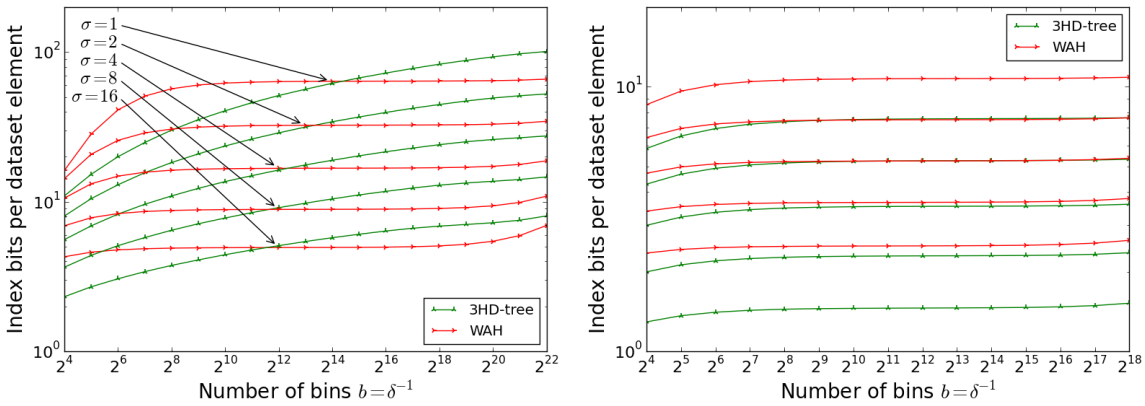
compression method must trade better compression in some cases for worse compression in others, with the goal being to higher compression in more common cases. Whether the bin counts for which HD-trees are advantageous are more common in practice depends on the use case; in our experience, however, many datasets fall into this intermediate range.

4.5.2 Zipf Bin Distribution, Equality Encoding

We now explore the effect of skew in the bin distribution on index size, an effect commonly observed in real-world datasets. For this purpose, we use a Zipfian bin distribution. If we let δ_i represent the density of the ith bin, a Zipfian bin distribution with parameter $s \in (1, \infty)$ yields $\delta_i \propto i^{-s}$. A greater parameter s leads to more pronounced skew. We use a Zipf distribution because it is a reasonable model for many datasets, it is easy to compute, and based on precedent [92].

Figure 4.3 shows the expected index size under two different Zipf parameters. Computing expected index size for a Zipf distribution is much more expensive than for the uniform distribution. This is because expected RSet size must be computed b times with b different δ values for b bins, instead of just once for uniform bin distribution since all bins have the same density. Thus, we have only charted up to $b = 2^16$ bins.

HD-trees perform relatively better under this skewed distribution versus the uniform distribution of Figure 4.2b. This is reasonable due to the weighted nature of storage impact across bins: though



(a) Uniform bin density distribution

(b) Zipfian bin density distribution, parameter $s = 2 \rightarrow \delta_i \propto i^{-2}$

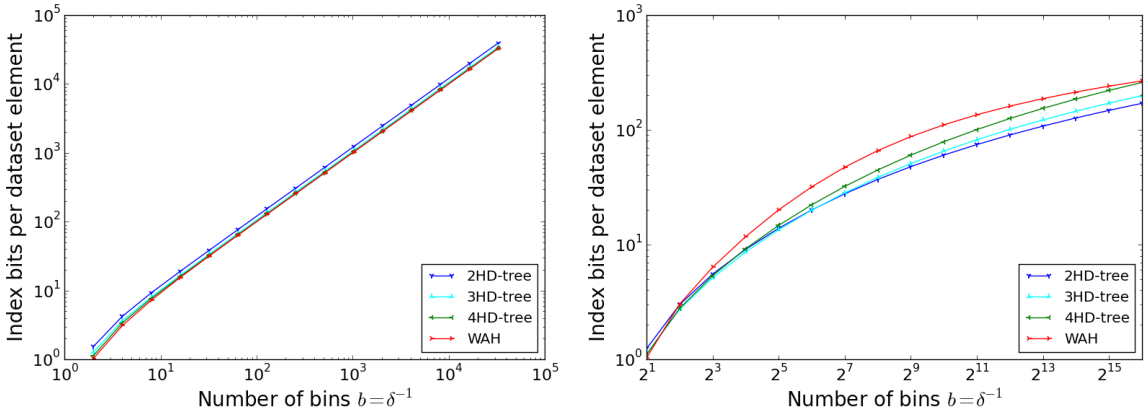
Figure 4.4 Comparison of HD-tree and WAH expected index sizes for *equality* encoding and varying streakiness (σ). Figures 4.4a and 4.4b show the relationship under a uniform and Zipfian bin distribution, respectively. In each plot, the curves of a given color represent increasing σ from top ($\sigma = 1$) to bottom ($\sigma = 16$).

WAH gives a lower bits-per-record ratio for the very low density bins, these contribute a relatively small fraction of the total index size, whereas HD-trees perform better on the bins with moderate density, which constitute the majority of total storage size. Many higher-density bins exist across a broad range of densities due to the skew, hence the consistently smaller index sizes under HD-trees in the presence of skew.

4.5.3 Effect of Clustering (σ)

We now HD-trees and WAH on RID sets with varying RID streakiness (σ). Figure 4.4 shows the effect of σ values from 1 to 16 in the context of a uniform and a Zipfian bin distribution, both with and equality encoding. The relative behavior of WAH and the HD-tree is generally maintained from Figures 4.2b and 4.3b are maintained, with the 3HD-tree still exhibiting the lesser storage footprint for intermediate bin counts in the uniform distribution case, and under all shown bin counts for the Zipfian case.

However, increasing σ does have an effect on the two methods, both in absolute and relative terms. In the uniform bin case, two effects are prominent. First, increasing σ reduces the crossover bin count at which WAH surpasses the 3HD-tree, implying that WAH handles higher clustering somewhat better than the 3HD-tree in this case. Second, it generally flattens both curves, thus reducing the impact of bin count on index size for both approaches. This flattening is not uniform



(a) Uniform bin density distribution

(b) Zipfian bin density distribution, parameter $s = 2 \rightarrow \delta_i \propto i^{-2}$

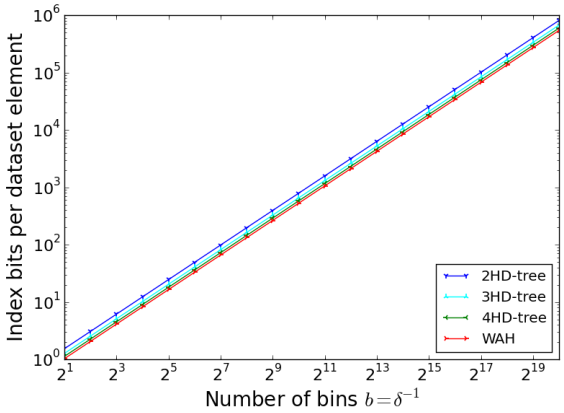
Figure 4.5 Comparison of expected index size under HD-tree and WAH RSet representations under a uniform bin distribution and *range* encoding, with no streakiness ($\sigma = 1$). Figures 4.5a and 4.5b show the relationship under a uniform and Zipfian bin distribution, respectively.

across both curves, however, leading to interesting changes in relative performance across both methods that would be an interesting subject for future exploration.

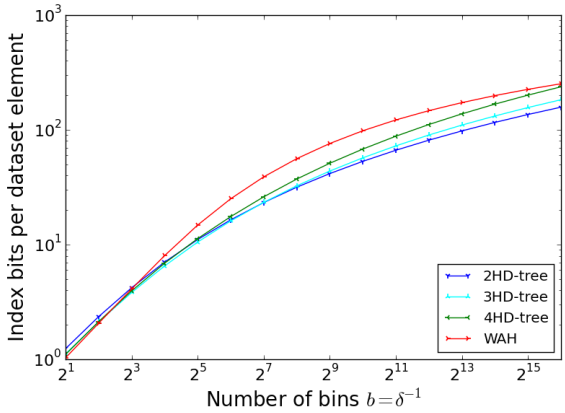
For the Zipfian case, the effect is different: larger σ values somewhat increase, not reduce, the gap between WAH and the 3HD-tree. A flattening effect is also observed, though it is more subtle.

We believe all of these effects derive from a tendency for higher σ values to make each bin behave as if it had a lower bin count (i.e., higher effective per-bin density δ_i). This is based on the intuition that a higher σ increases the chance of a run of sufficient length for compression under both WAH and HD-trees, an effect also achieved by a lower bin count (higher density δ_i). This is only a rough approximation, as increasing σ also has a flattening effect; however, it explains the trends seen here, with higher σ values appearing to *shift the curves left*.

In practice, the value of σ will vary across bins, especially in a skewed bin distribution (explored next). It is possible to measure σ in a given dataset or bin by deriving an n-gram table (specifically, 2-grams). Using this method, the authors have anecdotal found σ values near 1 to be most common, with more dense bins in highly-skewed datasets ranging up to 3 or 4. However, every dataset will differ in this respect; a more detailed survey of a range of datasets could shed some light on what typical σ parameters.

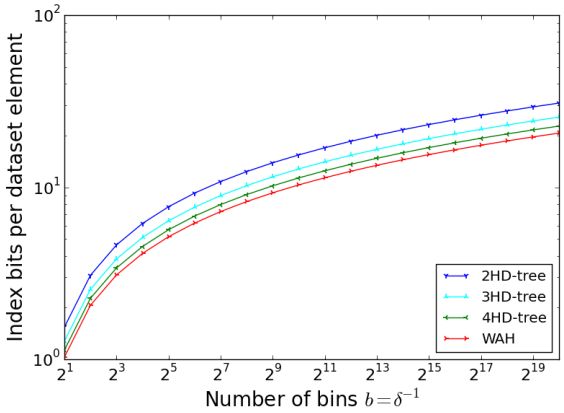


(a) Uniform bin density distribution

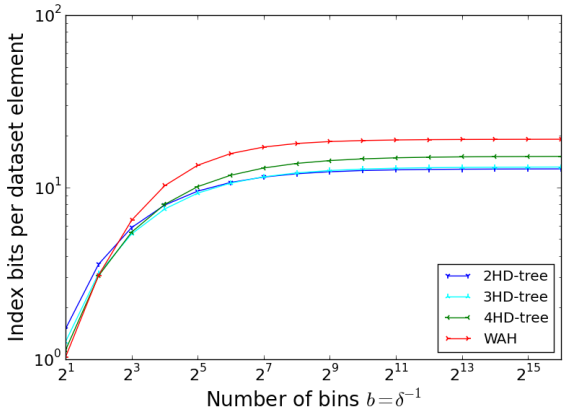


(b) Zipfian bin density distribution, parameter $s=2 \rightarrow \delta_i \propto i^{-2}$

Figure 4.6 Comparison of expected index size under HD-tree and WAH RSet representations under a uniform bin distribution and *interval* encoding, with no streakiness ($\sigma = 1$). Figures 4.6a and 4.6b show the relationship under a uniform and Zipfian bin distribution, respectively.



(a) Uniform bin density distribution



(b) Zipfian bin density distribution, parameter $s=2 \rightarrow \delta_i \propto i^{-2}$

Figure 4.7 Comparison of expected index size under HD-tree and WAH RSet representations under a uniform bin distribution and *binary component* encoding, with no streakiness ($\sigma = 1$). Figures 4.7a and 4.7b show the relationship under a uniform and Zipfian bin distribution, respectively.

4.5.4 Range, Interval, and Binary Component Encodings

Finally, we explore the impact of other common index encodings: range, interval, and binary component. As detailed in Section 4.2, index encodings cause individual bin RSets to be combined into a new set of *encoded RSets*. Each such encoded RSet is the union of some set of bin RSets, and there may be more or less encoded RSets than the original number of bins. The reader should reference Section 4.2 for exact definitions of range, interval, and binary component encodings.

Results for WAH versus 3HD-tree RSet representations under these three encodings, for both uniform and Zipfian bin distributions, are given in Figures 4.5, 4.6, and 4.7.

It is immediately apparent from these plots that WAH is superior under uniform bin distributions, at least slightly, but the 3HD-tree is generally smaller with a Zipfian distribution (parameter $s = 2.0$). Though the HD-tree has performed relatively better for Zipfian bins so far, the size of the jump in performance, as well as the fact that WAH is consistently better under uniform binning, is unusual. The root cause of this new trend lies in how these encodings differ from equality encoding.

Range, interval, and binary component encodings are related to one another, and different from equality encoding, in that their encoded RSets each incorporate a mean of half the bins. For range-encoded RSets, the number of bins combined via union is between 1 and $b-1$, with the mean being half the bins. Interval-encoded RSets each contain half the total number of bins ($\lfloor \frac{b}{2} \rfloor$, to be exact), while binary component-encoded RSetsconsist of about $\frac{b}{2}$ bins each (though this may be higher, especially for the "upper layers").

For a uniform bin distribution, this property implies that most/all encoded RSets have density about 0.5. We have seen in earlier plots (especially Figure 4.2b) that WAH uses significantly less storage than the HD-tree under these conditions, but with these alternative encodings, this scenario is commonplace rather than an unrealistic extreme. This result tells us that WAH is a better choice for use with these encodings under a uniform bin distribution (though uncompressed bitmaps might perform even better).

With a Zipfian bin distribution, however, the tables are turned. This is because the skew pushes densities away from 0.5 to intermediate densities more favorable to the HD-tree. These results indicates that, for more skewed data distributions, HD-tree may be the more compact index choice under these alternative encodings.

4.6 Conclusion

We have presented a theoretical performance model for expected index size under kHD-tree RSet representation. We compare this formula to those previous developed by Wu et al. [93] for WAH

bitmap compression, with slight modifications to the Markov model parameterization to improve the clarity of our experiments. Finally, we use these two models to explore the relative performance of WAH and HD-trees under a variety of scenarios, varying data cardinality, data skew, value clustering, and index encoding.

We conclude that WAH and HD-trees both have their strengths and weaknesses within the space of possible datasets. In our experiments, for equality encoding, we observe that HD-trees consistently perform better at intermediate data cardinalities (roughly, tens to tens of thousands of bins) and/or in the presence of Zipfian bin skew. Changes in value clustering ("streakiness" in our model terminology) does not have a substantial impact for uniform bin distribution, but favors the HD-tree when value skew is present. Finally, WAH generally performs better than the HD-tree for certain non-equality encodings (range, interval, and binary component in our tests) under a uniform bin distribution; however, the reverse is true given enough Zipfian bin distribution skew.

The number of possible experiments in this space is vast, and this study only scratches the surface. Not only do more bin distributions and index encodings exist to explore, but many more combinations of co-varying parameters are possible (for instance, what is the impact of changing σ under non-equality encodings?). Furthermore, there are very likely more meaningful parameters to explore than we have outlined, both in the data model itself (e.g., higher-order Markovian processes?) and in its application to the value-sliced index structure (e.g., varying σ by bin?). Therefore, while we have laid the groundwork for HD-tree storage performance modeling, and have derived some intuition as to what circumstances call for an HD-tree versus WAH index, there is definitely room for future exploration in this space.

Chapter 5

In Situ Data Transforms

5.1 Introduction

In response to the demands of increasing scientific simulation data scale, a variety of "data transformation" techniques have been developed. These transformations change the encoding of scientific data to achieve benefits including: reduced I/O costs and storage footprint via data compression [3, 13, 47, 52, 73], faster read access times via level-of-detail [40, 62] or storage layout optimization [32], and faster query response times via bitmap [88, 91] or inverted [41, 42] indexing. Also to cope with increasing data scale, many of these data transforms have been applied *in situ* by co-locating with the application on compute cores or *in transit* by utilizing dedicated staging nodes/cores, thus moving away from data *post*-processing to data *co*-processing.

Despite offering numerous potential benefits, actual adoption of in situ data transforms by scientific applications has been inhibited by several obstacles. First and foremost among these is the yet-unresolved key question of *where to place data transformation services within the end-to-end scientific workflow*. This question has been a subject of much debate in the scientific community at large. The consensus, especially among end users, seems to be emerging in favor placement at the *I/O middleware level*. For instance, different scientists working on the exascale GTS fusion plasma and S3D combustion simulation codes have stated that using data compression or indexing through I/O middleware such as ADIOS makes the most sense for them[1].

In response to this critical need, this paper aims to bridge the growing gap between the increasing number of data transformation methods and the limited rate of their transparent integration within scientific workflows. Specifically, the paper makes the following **contributions**:

[1] stated by application scientists at an SDAV conference call

1. We design a **generic framework** for transparent, in situ data transformation in the ADIOS I/O middleware (see Section 5.3 for the rationale behind selecting ADIOS as the platform). This framework is now included in the ADIOS 1.6.0 release.

2. We reduce the *cost of integration* of data transformation services within scientific applications and avoid *data semantics erasure* (i.e., the degeneration of structured data into simple byte arrays when transforms are applied outside of I/O middleware) by ensuring **user transparency** of the proposed framework within ADIOS, thus leveraging existing I/O middleware integration and maintaining the capture of data semantics.

3. We enable *ease of transform experimentation, tuning, and swapping* through **runtime configuration of data transform plugins**. By augmenting the ADIOS XML runtime interface, as opposed to using a hard-coded configuration, transforms can be selected and tuned without modification or recompilation of application code.

4. We support *I/O pipeline flexibility/compatibility* by explicitly **decoupling data transformations and I/O transports** in ADIOS, thus achieving orthogonality between these services. This permits users to continue using existing I/O transports while augmenting their I/O pipeline with data transforms.

5. We address the *missing support for read-optimizing transforms in I/O middleware* by incorporating a **flexible read model** into our transforms framework.

6. We have integrated and evaluated a variety of data transformation methods in the categories of compression, level-of-detail, indexing, and storage layout optimization. While only some appear in our results due to space constraints, the possibility of implementing this range of methods underscores the flexibility of the framework. Code for these plugins, along with the transform framework codebase used in this paper, is also included in ADIOS version 1.6.0 and later, the source code for which may be found on the Oak Ridge National Lab website [58] or GitHub [59] at the time of this writing (some plugins require additional libraries [69]).

Using this framework, we also evaluate the benefits and drawbacks of data transformations in general for both writes and reads (see Section 5.5.3). On the write side, we apply multiple data transformations to the existing Quantum Lattice Gas [83, 84] (QLG) simulator on up to 8,192 cores on the leadership-class Titan supercomputer at Oak Ridge National Lab (without modifying the application code), demonstrating the framework's scalability, as well as the potential for in situ compression to reduce I/O time and storage footprint. On the read side, we analyze the performance of different read access patterns over transformed data. We show the impact of "transform opacity"

(i.e., the atomicity of transformed data), and how our flexible read model enables some transforms to overcome this effect to improve I/O time, even outperforming the no-transform case.

5.2 Related Work

Many modern scientific applications use publicly available I/O middleware solutions, such as HDF5 [28], ADIOS [54], and pnetCDF [49] to handle complex parallel I/O tasks in HPC systems. Such libraries provide stable and portable I/O interfaces, and store data in self-describing file formats that form the underpinnings of many scientific data workflows. However, there is limited support for run-time data transformations.

For example, HDF5 offers end-users the ability to incorporate custom filters such as run-time data compression [99]. However, the lack of parallel write support for transforms makes it inapplicable for in situ use at scale, and only hard-coded transform configuration is available (limitations we address in our ADIOS-based approach).

The more recent Damaris [27] middleware uses a dynamic plugin-loading architecture that can support some data transforms (compression has been demonstrated). However, despite its flexibility, the shared whiteboard model employed does not in itself overcome data semantics erasure, as it layers data transforms above the I/O layer, unlike the approach proposed in this paper.

ADIOS [54] features a modular I/O transport layer enabling runtime selection of I/O methods. Attempts to use this layer for data transformations [1, 2, 103], while yielding useful research results, inherently prevent the user from choosing any other I/O transport. We address this issue through I/O decoupling.

Finally, many standalone *read-optimizing data transforms* have been demonstrated, including level-of-detail encoding [40, 62] and storage layout optimization [31, 32]. However, to our knowledge, no current I/O middleware is capable of offering these transforms, along with their resultant I/O-reduction benefits, to end users as a service.

On the subject of data service placement, besides *in situ* placement, movement of data to staging cores or nodes for processing, *in transit* placement, has also been explored in the literature. For example, DataStager [1] moves data asynchronously from compute nodes to staging nodes, enabling data processing in the staging area. PreDatA [103], an extension of DataStager, characterizes and reorganizes application data to speed up data analytics by offering both in situ and in transit placement options. JITStager [2] provides a more dynamic infrastructure to customize data movement at scale by leveraging both compute and staging nodes.

While in transit placement has proven to be an effective strategy, here we focus on in situ placement of data transforms for two main reasons. First, in situ placement makes more sense as a

first step, because achieving in transit placement is then relatively straightforward. For instance, the `DATASPACES` and `DIMES` [29] staging I/O transports send data to staging cores/nodes, where those data may then be written to disk via a second call to ADIOS. By applying "in situ" data transforms at the staging level, we effectively achieve "in transit" data transforms without further effort. In contrast, porting an in transit-only solution back to in situ may be obstructed by implicit dependencies, such as assumed exclusive ownership of staging cores, etc. Second, in transit placement relies on the availability of staging nodes, whereas in situ does not, making the latter more widely applicable at this time. In the future, however, a native in transit data transform framework could be developed in ADIOS as an extension of this work.

5.3 Background: ADIOS

We choose to base our transform framework in the ADIOS I/O middleware [54] for a number of reasons, besides our hands-on experience with its internals. First, ADIOS has a broad user-base, with more than a dozen large-scale applications using ADIOS in production. Adoption of various data transformation services through our framework by these applications could provide invaluable feedback on the design principles that underlie our framework and suggest ways to improve its functionality. Second, ADIOS already implements a *modular I/O transport framework*, which we require for our goal of decoupling I/O and data transforms. Finally, the "process group" data parallelism model of ADIOS simplifies our parallel implementation.

Next, we review key technical components of ADIOS pertinent to our work.

5.3.1 Modular Layered Service Model

ADIOS is divided into three main layers: write and read "transport" layers, and a "common" layer. The transport layers handle input and output from storage, respectively, and are modular, allowing for read and write I/O transport technologies (called "transport methods") to be selected independently at run-time. In this paper, we often group these as simply the "transport layer." In contrast, the common layer acts as the central "hub" of ADIOS, from which the transport layer is invoked. The common layer defines the API for transport methods, as well as the top-level user API.

Here, we develop a new layer, the "**data transformation**" layer, analogous to the transport layer, to support the modular integration of data transformation methods.

5.3.2 Process Group Data Model

The fundamental unit of data in ADIOS is the "Process Group," or "PG." A PG contains all data produced by a single writing process during a single write step (e.g., 16 writing processes and 4 write steps yield 64 PGs). This layout is depicted in Figure 5.1. Each PG contains one or more variables, each defined as a scalar or a multidimensional array of a certain datatype. A variable may be defined across many PGs, with each PG containing a local "block" of that variable, together comprising a logical global array. The PG model achieves high write throughput by eliminating data exchange among processes [53], while also performing well for reads [63].

When reading data, ADIOS abstracts away the division of data into PGs. The transport layer transparently converts read requests in the global coordinate space to a set of PG-local read requests, then stitches the results together before returning them to the user. While this translation is convenient to the user, the fact that it is handled in the transport layer presents a challenge to data transformation, as explained in Section 5.4.3.

Figure 5.1 The PG data layout in an ADIOS file. Variables are partitioned by PG's according to which process produced which chunk.

5.3.3 Runtime Configuration via XML

To support runtime configuration, ADIOS uses an XML configuration file that is parsed during initialization. It includes the schema for variable types and dimensions, as well as the write transport method to use (along with parameters). We extend this configuration file to enable tagging of transform methods on variables (see Section 5.4.1).

5.3.4 ADIOS Read API

As our flexible read model is based on selectively reading transformed data based on user read requests, it is important to first describe the read semantics of ADIOS. ADIOS supports various read "selections." These currently include "bounding box" (i.e., subvolume) and "point list" selection types, which operate in the global coordinate space, and "writeblock," which reads all of a variable's

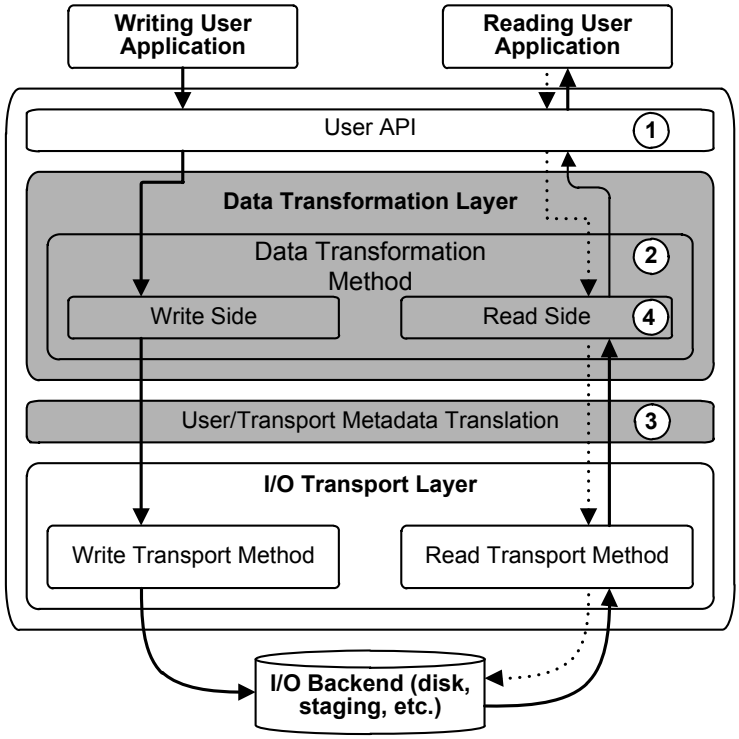


Figure 5.2 An overview of the transforms framework within ADIOS. The shaded area is newly-added to enable data transformation, while the number markings roughly indicate where each of our four approaches are implemented (1: user transparency, 2: runtime configuration, 3: I/O decoupling, and 4: flexible read model). Solid arrows indicate data movement, while dotted arrows indicate read requests.

data from a certain PG. ADIOS also supports two return policies: user buffer mode and chunking mode. For the former, the user supplies a sufficiently-sized buffer for ADIOS to populate with the entire result. For the latter, ADIOS returns pieces of the result ("chunks") over time; this mode is designed for memory-constrained and streaming environments.

Pursuant to user-transparency, we endeavor to support these access methods over transformed data (see Section 5.4.2). Also, our flexible read model for supporting read-optimized transforms interacts with read requests from this API to decrease I/O time (see Section 5.4.3).

5.4 Method

Figure 5.2 presents a high-level overview of our transforms framework within ADIOS. The data transformation layer (the shaded region) provides the framework for incorporating transforms

into the ADIOS data flow. Similarly to the I/O transport layer, the data transform layer defines a standardized interface for integrating new data transformation "plugins" into ADIOS. Each plugins is divided into a write and read component, as our flexible read model requires differentiated functionality from the write side (see Section 5.4.3). Plugins themselves are written in C (as is ADIOS), but their transforms services are also compatible with ADIOS's other language bindings (e.g., FORTRAN and C++).

The transforms framework consists of three main components:

- **Runtime-configured plugins** (Section 5.4.1) allow for different data transforms to be selected/configured by the user, with the plugins themselves defining the actual data transformation.
- **User transparency** and **I/O decoupling** (Section 5.4.2) ensure that the data transform layer is compatible with ADIOS, both with the User API above and the I/O transport layer beneath.
- **Flexible read model** (Section 5.4.3) explains how read-optimizing transforms can be empowered to influence I/O and to speed up read times.

5.4.1 Runtime-configured Plugins

Traditionally, adding data transform services to a scientific application requires *ad hoc* integration with its tested/stable codebase, a risky and time-consuming proposition. Runtime configuration of data transforms through an I/O middleware can reduce such costs and provide a mechanism for transform experimentation, tuning, and swapping, as changes will not require application code modification/recompilation. It can also avoid "technology lock-in," which leads to the risk of dependence on particular data transform codes that may become unsupported in the future.

```
<var name="temperature"
type="double"
   dimensions="NX,NY"
transform="zlib:5"/>
```

Figure 5.3 An example ADIOS XML configuration defining a variable, with our specification of a data transformation added (zlib at compression level 5, in this case).

The runtime configuration of data transforms plays a simple, yet important, role in the data transform framework: it is the only component that an end-user directly interacts with. The interface is an extension of the existing ADIOS XML configuration for defining data variables (Figure 5.3). The

added attribute, "transform," specifies both the specific data transform to apply (zlib compression, in this case) as well as any parameters that transform accepts (here, the zlib speed vs. quality parameter).

Internally, the user's install of ADIOS is configured with a set of transform plugins, each of which defines the functions necessary for encoding and decoding data passing through the I/O transport layer. The transform name specified in the XML is mapped through a table to the corresponding plugin code, and the parameters after the colon (':') are passed along. This XML is only needed during the write process; when reading, any data transforms that were applied to the data were also recorded alongside it, and so can be detected automatically.

5.4.2 User-transparency and I/O Decoupling

One of the key features of the I/O middleware packages used by scientific applications is portable, self-describing file formats, which form the underpinnings of many scientific data workflows. Handling the encoding/decoding of a data transform (e.g., compression) at the application level inherently strips the semantics from the data (datatype, grid dimensions, etc.) by converting it into an opaque byte array, thus distorting the original form of the data, and, as a result, countermanding the self-describing file format and breaking any associated workflow.

Placing the data transform layer within ADIOS overcomes data semantics erasure, but introduces challenges of its own. By virtue of its location in the ADIOS stack, the data transform layer must now maintain compatibility with both the *User API* above and the *I/O Transport layer* below (as depicted in Figure 5.2). Additionally, this compatibility must be maintained during both the writes and reads. Here, we cover the write side only, as the read side is more naturally explored in context with the flexible read model (Section 5.4.3).

Achieving transparency on the write side can be reduced to achieving *physical data independence*, that is, translating between the *logical* and *physical* view of the data. Currently in ADIOS, the user will provide variable data with associated *logical* metadata (e.g., tagging a variable as a 3D array of double-precision values). Then, this same data and metadata are passed to the I/O transport layer, which assumes that they match the physical representation on disk (e.g., data on disk is a contiguous chunk of product-of-dimensions times size-of-datatype bytes).

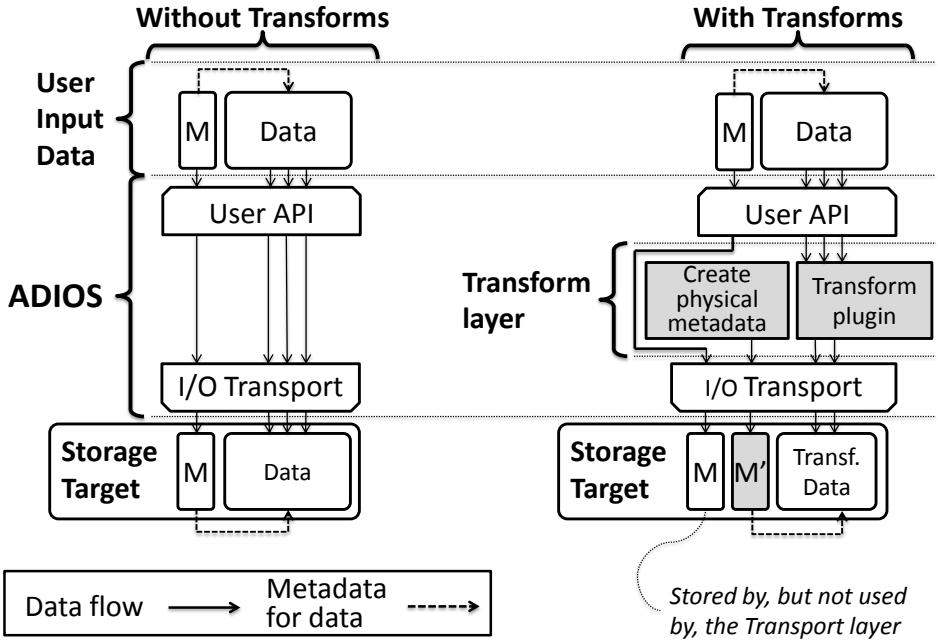
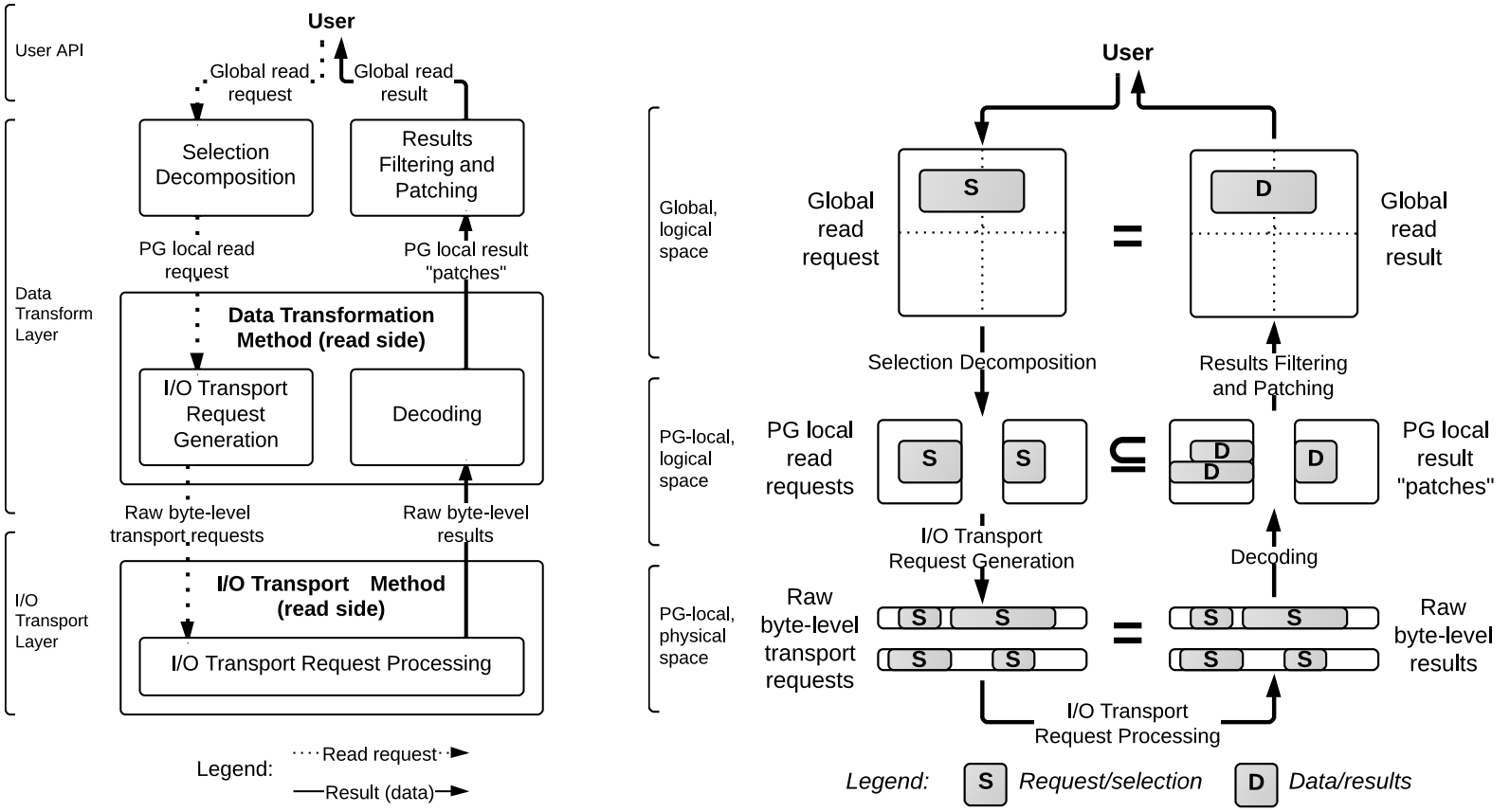


Figure 5.4 A comparison of writing with and without a data transform. In both cases, the user passes the variable's data along with its corresponding metadata (dimensions and datatype, denoted with an "M" in the figure). Normally, ADIOS uses the user-provided metadata to determine the data layout in storage; however, when a data transform is selected, metadata translation generates a new physical metadata and presents that to the transport layer, instead. The original metadata is stored for later use.



(a) Read Process - Component View

(b) Read Process - Dataflow View

Figure 5.5 The read process for a transformed variable, illustrated through a dual perspective. Figure 5.5a shows the *component view*, with boxes representing the *actions* taken to complete a read request, whereas Figure 5.5b shows the *dataflow view*, highlighting the evolution of the *read request itself*, and later the corresponding result data, as it flows through ADIOS. The figures are complementary: the objects in the dataflow view are the transitions (inputs/outputs) between the actions in the component view and, conversely, the actions in the component view correspond to transitions converting between requests/results in the dataflow view.

We break this equivalence to permit a physical representation different from the logical one through *user/transport metadata translation*, as depicted in Figure 5.4. Two copies of the dimension/datatype metadata are maintained for each transformed variable: a logical and a physical version. The logical metadata is what is received from, and later presented back to, the user. In contrast, the physical metadata is a fabricated 1-dimensional byte array for each transformed variable, with length equal to the post-transform size. The I/O transport layer is only exposed to the physical metadata, effectively "tricking it" into treating the transformed data as a byte array. The logical (original) metadata is *stored* by the transform layer for later use, but is never *interpreted* by the I/O transport layer.

This metadata translation process operates alongside of, and in support of, the *data* translation (i.e., transform). The actual data transform is performed by the specific transform plugin, through the main Write API function, "apply," which takes the original data buffer and produces a new, encoded buffer. More detail on this API is available via the source code on GitHub [59].

Thus, under this strategy, the user always sees the original form of the data, whereas the I/O transport layer receives the transformed data as if the user had written directly as 1D byte arrays, a supported use case in ADIOS. The transform layer plays "middle man," presenting to the user and I/O transport layer their corresponding views of the data, thus achieving both *user transparency* and *I/O decoupling*.

5.4.3 Flexible Read Model

We begin with a motivating example for our proposed "flexible read model." Consider the APLOD level-of-detail encoding [40], which optimizes read performance by supporting partial-precision reads with multi-fold I/O reduction *and* full-precision reads with little overhead, both on the same copy of scientific data and with no storage overhead. This encoding (a data transform) can achieve I/O speedup by reading a subset of the data from disk via out-of-core optimization. However, this optimization is nullified if entire chunks of APLOD-encoded data are read from disk by an I/O middleware before passing them to APLOD, as it is too late to achieve I/O reduction after the chunk is read. Instead, this reduction must occur earlier by consulting the specific transform plugin (APLOD, in this case) *during the read scheduling phase* to eliminate unnecessary I/O.

Thus, to fully support read-optimizing transforms, such as APLOD, we propose a protocol for modified read scheduling, herein referred to as the "flexible read model." The read pipeline implementing this model is illustrated in Figure 5.5 through a dual perspective: the *component view* and the *dataflow view*. Figure 5.5a shows the sequence of *actions* that take place to answer a user's read request, while Figure 5.5b depicts the evolution of the *user's request itself*, and later

the returned data, as it flows through the pipeline. Each component of this pipeline is described separately in the following subsections.

It should be noted that the read pipeline complexity stems not only from integrating "black-box" transform plugin input into read scheduling, but also from the PG model espoused by ADIOS. ADIOS's PG model induces a "reader-makes-right" model that involves data patching/filtering procedures to abstract away the division of a file into PGs. Likewise, though the PG model simplifies the transform write side, the read side is complicated by the need to handle this patching/filtering, as well.

It is helpful to consider the different *coordinate spaces* at play throughout this data pipeline. Initially, user selections exist in the *global*, *logical* space, that is, with absolute, global coordinates across all PGs defined in terms of the data's logical form (e.g., 3D array of double-precision values). In contrast, due to *metadata translation* (Section 5.4.2), the I/O transport layer services requests in the *PG-local*, *physical* space, that is, relative to a given PG and in terms of byte offsets within that PG. Thus, the read process essentially consists of translation of user requests down to the PG-local, physical space, followed by reverse translation of the result data back to the user's global, logical space.

Throughout this process, the transform plugin remains a "black box," so we first discuss the plugin read API we have developed, which defines the "tools" each transform plugin makes available to the framework. After this, we break down each step of the read process in Figure 5.5a.

5.4.3.1 Plugin Read API Overview

The functions of our plugin read API are listed in Figure 5.6. The first function, `generate_raw_read_requests`, services the "I/O Transport Request Generation" step in Figure 5.5a, whereas the other functions support the "Decoding" step.

The `generate_raw_read_requests` function is where the transform plugin inserts its guidance into the read scheduling process: given the portion of the user's selection that overlaps a certain PG (i.e., a PG-local, logical selection), this function must produce a series of PG-local, physical read requests to answer that request. Later on, the other three functions (i.e., `*_request_completed`) are called with the bytes read, and are expected to produce chunks of logical data from them.

The "request" parameter types correspond to the three levels of requests shown in Figure 5.5b, from top to bottom, and contain information specific to that level:

- `global_read_request` is the original read selection;
- `pg_read_request` describes a PG-local portion of the original read selection; and

```
// For I/O Transport Request Generation
void generate_raw_read_requests(
             global_read_request *g_read,
             pg_read_request *pg_read);

// For Decoding
datablock * raw_read_request_completed(
             global_read_request *g_read,
             pg_read_request *pg_read,
             raw_read_request *completed_raw_read);

datablock * pg_read_request_completed(
             global_read_request *g_read,
             pg_read_request *completed_pg_read);

datablock * read_request_completed(
             global_read_request *completed_g_read);
```

Figure 5.6 An outline of our ADIOS data transformation plugin read API

- `raw_read_request` is a byte-range read request from storage.

There is a clear hierarchy among these types: each global request begets one or more PG requests, and each PG request begets one or more raw requests. In fact, these types are linked in a tree structure, and each callback receives a "path" in this tree from the root (global request).

The other datatype listed, `datablock`, represents a chunk of data in the global, logical space to facilitate physical-to-logical data mapping. Each `datablock` contains a buffer, a datatype, and a selection object defining its bounds/shape.

5.4.3.2 Selection Decomposition

The first step in completing a read request is to generate a set of PG-local selections from the user's global selection. While ADIOS's transport layer already does this for non-transformed variables, due to user/transport layer metadata translation, the transport layer no longer sees the logical view of the data, so we replicate this process in the transform layer. Currently, we use ADIOS's algorithm: intersect the user's selection with the bounds of each PG. This performs adequately at current file scales, though in the future a more efficient intersection algorithm (e.g., using space partitioning) could be added.

5.4.3.3 I/O Request Generation

The next step is to translate the PG-local selections into physical (byte-level) reads. As discussed at the top of Section 5.4.3, the straightforward approach of simply reading the entire PG to be passed through the transform plugin cannot exploit read-optimizing transforms. Thus, we take a *plugin-driven* approach, where the transform plugin itself dictates which portions of the transformed data are required. The `generate_raw_read_requests` function allows the plugin to drive this process by emitting raw read requests based on each PG-local portion of the user's selection.

It's important to note that, even with this flexibility, plugins that do not require flexible read support (e.g., compression) are not complicated. In such cases, any touched PG can simply be scheduled for read in its entirety, requiring only two lines of code, and during the later de-transformation step, the decoded result can be returned as a single datablock with one more line of code.

5.4.3.4 Decoding

Once the raw read requests scheduled by the transform plugin begin to complete, the results are passed back to the plugin via the `*_completed` callback functions in the API (Figure 5.6). `raw_read_request_completed` is called for each completed raw read, `pg_read_request_completed` is called when all raw reads associated with a PG-local request are done, and `global_read_request_completed` is called when all raw reads stemming from a user's original read request are finished. Results may be returned from any of these callbacks, giving the plugin flexibility in reconstructing the requested data. For instance, a plugin may wish to return results immediately upon receiving each chunk of raw bytes it requested. Alternatively, it may wish to accumulate raw reads until an entire PG's worth are collected, at which time it may return a single, larger result.

Note that, for simplicity of plugin implementation, plugins may return whatever chunks of data are convenient, even if they contain extraneous data, so long as the sum total fully covers the user's selection. Data unrelated to the user's request are filtered out in the next step.

5.4.3.5 Results Filtering and Patching

As `datablocks` are produced by the transform plugin, their contents must be applied to user's results. As shown by the *PG-local result "patches"* in Figure 5.5b, there may be many `datablocks` returned, which may include data outside of the original selection. This process is handled by a set of optimized "patching" functions within the framework, which take as input a source `datablock` (from the plugin), a target `datablock` (the user's results buffer), and the intersection region. Each `datablock` may contain a selection of a different type (bounding box or point list), so an optimized version of the patching function is required for each combination.

Once all `datablocks` have been produced by the transform plugin and properly patched into the results, the user's request is completed (i.e., the user's result buffer is full or all chunks have been returned, depending on result mode).

5.5 Results

Our primary contribution, adding data transformation services to ADIOS, is demonstrated by successful use of data transforms in ADIOS, which includes the experiments that follow. The explanations accompanying these tests attempt to point out how specific goals, such as runtime configuration and user transparency, are realized in practice.

Additionally, in the course of building and testing the transform framework, we have added several data transform technologies as plugins, including:

- zlib [64], bzip2 [74], szip [99], ISOBAR [73] (compression)
- APLOD [40] (level-of-detail)
- ALACRITY [41] (indexing)
- PARLO [32] (layout optimization)

While only some of these appear in our evaluations below for brevity, this range of plugins gives an indication of the flexibility of the data transform framework. Note that the transform framework is included in ADIOS versions 1.6.0 and later; the source code may be found on the ORNL website [58] or on GitHub [59] at the time of this writing, with additional library code needed by some of the transforms found elsewhere [69].

However, some aspects of the data transform framework do lend themselves to quantitative analysis. Specifically, the measurement of overhead induced by the transform framework itself (writes and reads), the behavior of reads over transformed data, and the effect of read-optimizing transforms, are each examined in the following subsections. First, though, we describe our experimental setup.

5.5.1 Experimental Setup

The write-related tests are performed on the Titan supercomputer at Oak Ridge National Lab. Titan consists of 18,688 16-core compute nodes (299,008 cores), with 32GB of RAM per node. Additionally, each node houses an nVidia Tesla K20 GPU with 6GB of RAM, although these are not used in our

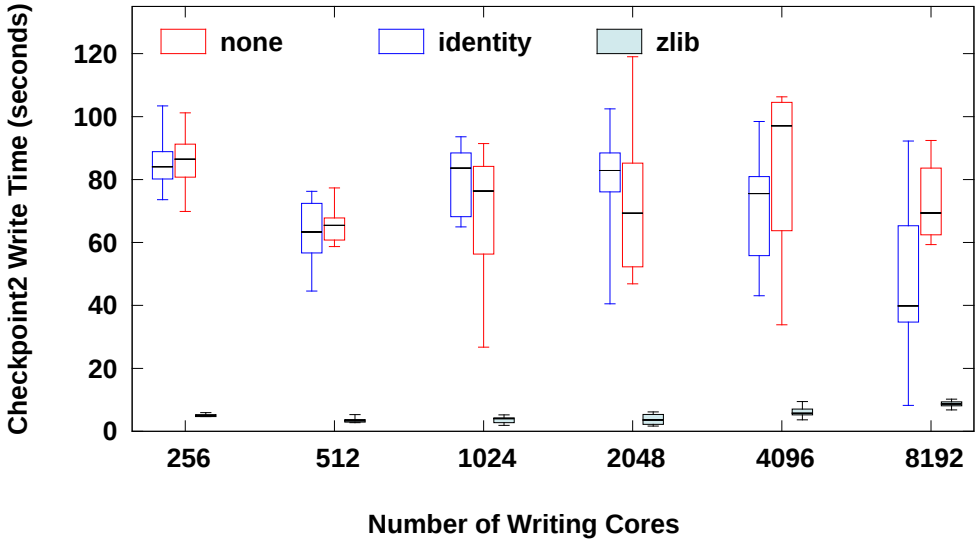


Figure 5.7 Total I/O time for a QLG simulation checkpoint operation when using different transforms. In this box-and-whiskers plot, the box spans the interquartile range, the line within the box sits at the median, and the "whiskers" extend to the min and max timings.

experiments. To provide a real-world scenario for write testing, we run the large-scale Quantum Lattice Gas (QLG) simulator [83, 84], which uses ADIOS for its checkpointing process.

The read-related tests are performed on the Sith development cluster at ORNL. We use data from a large-scale S3D combustion simulation run. The whole dataset is roughly 22TB; however, varying the PG size and the data transforms in our experiments requires maintaining many copies of the data, which is untenable at this scale. Therefore, we extract and use the first 4 timesteps of the "temp" temperature variable, a subset of approximately 50GB. Due to the partitioning of variables and timesteps in ADIOS files, we believe performance on this subset to be indicative of that for cases with more variables and/or timesteps.

All experiments utilized the shared Lustre parallel filesystem, with default Lustre configuration parameters (4 OST stripe count, 1MB stripe size). Also of note, all experiments were performed under the "Widow" Lustre filesystem, prior to the upgrade to "Atlas" in late 2013.

5.5.2 Framework Overhead (Write and Read)

In practice, evaluating the overhead of a transform framework itself is non-trivial, as any transform plugin used will necessarily contribute to the total run time. Therefore, we have developed a special

Table 5.1 Storage footprint of checkpoint files under different data transforms. File sizes are given in gigabytes.

Cores	256	512	1024	2048	4096	8192
none	61.04	61.04	61.04	61.05	61.08	61.15
identity	61.04	61.04	61.04	61.05	61.08	61.16
zlib	0.96	0.97	0.94	1.55	1.58	1.59

"**identity**" transform plugin, which does not modify data passed through it (thus inducing negligible overhead, yet fully exercising the transform framework by virtue of being treated as a "black box." For fairness, the identity transform leverages our flexible read model to enact the same data sieving policy used by the read transport layer on non-transformed data.

Our first experiment evaluates write performance with a large-scale run of the QLG simulation, which uses ADIOS to write checkpoint files. We vary the number of cores used between 256 and 8,192, and evaluate with transforms "none" (no transform) and "identity", as well as the "zlib" compression plugin to show that realistic transforms work at scale. QLG is configured to write approximately 61GB per checkpoint, a total data size that is kept constant as the core count varies; i.e., strong scaling is used.

Note that *no modifications to the simulation code were required* to apply data transforms in this experiment, and different transforms were selected via a one-line edit in the ADIOS XML file. Also, with respect to I/O decoupling, the results below are based on using the `MPI_LUSTRE` transport method (a collective MPI-IO method with data-alignment algorithms tuned for Lustre), but we have also successfully run this application with the `MPI` and `MPI_AGGREGATE` I/O methods, again without any application code changes.

The checkpoint I/O timings are shown in Figure 5.7. Each data point is derived from 9 repetitions in order to give a more statistically-significant result. During our tests, the `MPI_LUSTRE` method seemed to experience one-time overhead at the first write operation (with or without transforms enabled), which would have skewed our comparison. Since our interest is in evaluating transform overhead, not I/O transport performance, we simply use the second checkpoint timings instead, which do not exhibit this effect. We use the quartile-based box-and-whiskers plot format due to the I/O variability at this scale. In the plot, each box represents the interquartile range (25th to 75th percentile), with a line at the median, and a "whisker" at the min and max timings.

A few trends are evident in these plots. For lower core counts, write time is quite consistent between "none" and "identity," implying no discernible overhead. Variability increases for larger core counts (as expected, since there is more room for outliers/stragglers), making the comparison

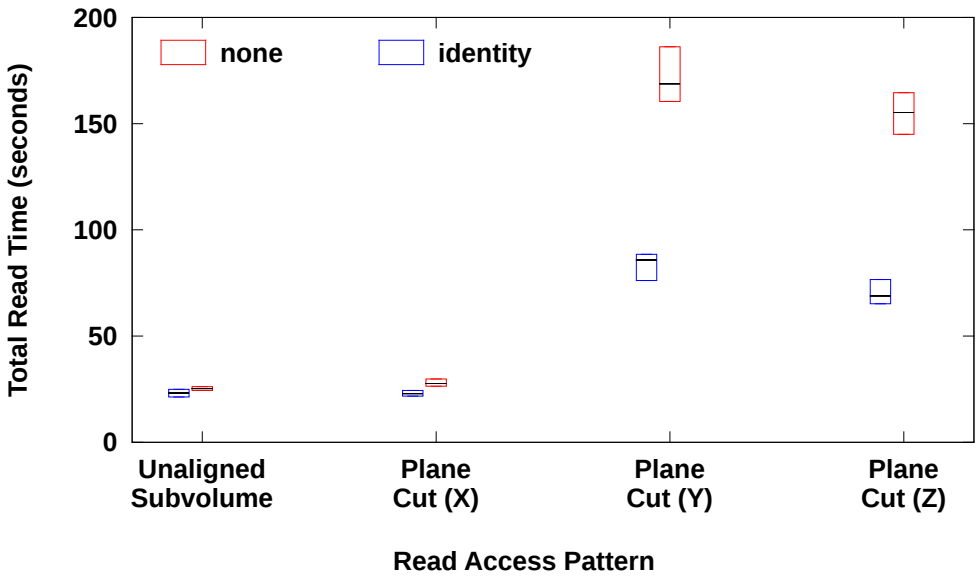


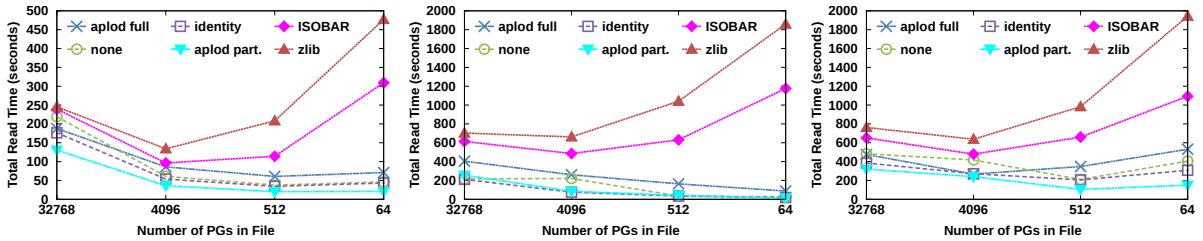
Figure 5.8 Total read time over S3D data under different transforms. In this box-and-whiskers plot, the box spans the interquartile range and the line within the box sits at the median.

less clear, but there is no clear indication of overhead with identity. The "zlib" transform, in contrast, exhibits much lower and more consistent write times than none/identity. This is due to the highly-compressible nature of the QLG data (as evident in Table 5.1), which greatly reduces I/O. While other scientific data are typically less compressible, there is still the potential for I/O reduction to outweigh compression time in some cases.

To test overhead on the read side, we construct another experiment as follows. Starting from S3D combustion data, we generate a no transform ("none") and "identity" transform copy, then compare their performance under different read access patterns. In order to test parallelism on the read side, all tests are performed using 64 reader cores.

The read performance results are given in Figure 5.8. The test for each datapoint includes 500 randomly-generated read selections (with the same random seed being used for each transform type to ensure fairness). The entire experiment was repeated 30 times, with the interquartile range and median for each configuration plotted.

We anticipate similar performance between "none" and "identity," yet the results show "identity" outperforms "none" in the Y/Z plane cases. Both "identity" and "none" induce the same number of seeks and bytes read, and submit I/O requests in the same order. The experimental results were consistent across multiple reruns on different days, with and without the same Lustre OSTs enforced



(a) Subvolume access pattern **(b) X plane cut** access pattern **(c) Y plane cut** access pattern

Figure 5.9 Read performance over S3D data under different access patterns and transform methods. "aplod full" and "aplod part" refer to APLOD in full-precision and partial-precision read modes, respectively.

for both "none" and "identity" files, and with careful attention to eliminating cross-test caching effects (various parameter iteration orders were tested). Total CPU time is roughly the same (at most 1-2% overhead relative to total read time for "identity").

Having ruled out obvious causes, one possible explanation is that, while both "none" and "identity" perform similar data selection computations, the distribution relative to I/O calls differs. While "none" performs these computations *within* the I/O loop, "identity" invokes a different codepath that performs them as pre-/post-processing steps, with a tight, relatively-computeless I/O loop in between. We believe this difference causes "identity" to experience better filesystem caching and responsiveness due to rapidity of I/O requests. However, users would be advised not to try using "identity" in the hopes of increased read performance.

5.5.3 Read Performance of Transformed Data

In any general data transform framework within an I/O middleware, there must enter the concept of "chunking": the global dataspace is partitioned into "chunks" in some fashion, and each chunk is encoded separately by the transform. The only I/O middleware known to the authors with any form of built-in data transform support, HDF5 with its "filters", follows this rule, as do we here.

Based on this general principle, we can make some inferences about read performance over data transformed. For instance, in general, increasing chunk sizes in a file improves the performance of most access patterns by reducing seeks. However, for data transforms that are "opaque" (i.e., where a chunk subset cannot be retrieved without reading/decoding the entire chunk), we expect a counteracting performance penalty as chunk size increases. This implies a tipping point, where the benefit of larger chunks due to larger contiguous access is outweighed by the penalty of reading more and more irrelevant data. In contrast, transforms that are "transparent" (i.e., where a chunk subset may be retrieved by reading/decoding *less* than the entire chunk), this performance penalty

is reduced or eliminated, thus propagating the benefit of larger chunks.

Read access patterns also play a role here. Depending on the underlying linearization of the data, some access patterns (such as plane cuts perpendicular to the slowest-varying dimension) yield far more contiguous access than others (such as plane cuts on another dimension). For non-transformed and transparently-transformed data underpinned by such a linearization, non-contiguous access patterns will degrade performance at larger chunk sizes, because the number of seeks between non-contiguous segments of pertinent data will increase. This effect does not occur for opaque transforms, which already read/decode each touched chunk in its entirety, regardless of access pattern. Thus, the performance gap between transparent and opaque transforms should shrink for less-contiguous access patterns.

Based on these conjectures, we devise an experiment to vary access patterns and vary chunk sizes (i.e., in ADIOS, PG sizes) for different transforms. We perform this test with three access patterns: subvolume, plane X (contiguous), plane Y (non-contiguous); four PG subdivisions of the global space: 2^3, 4^3, 8^3, and 16^3 chunks; and six transforms: none, identity, zlib compression, ISOBAR compression, and APLOD level-of-detail at two precision levels (partial and full precision). Each experiment run consists of 50 reads of a given access pattern, and we perform 10 repetitions of each experiment configuration and average the results.

Among these six transforms, four are considered to be "transparent" (none , identity, and both APLOD modes) and two "opaque" (zlib and ISOBAR, both being whole-PG compression techniques).

The results are shown in Figures 5.9a, 5.9b, and 5.9c. The trends inferred above can be observed in action, as well as a few other insights:

1. None and identity perform very similarly, implying negligible overhead by the transform framework.

2. The transparent transforms (none, identity, and APLOD) show improved performance with larger chunk size on mostly-contiguous access patterns (subvolume and plane X), but less so for non-contiguous access patterns (plane Y).

3. For the opaque transforms (zlib, isobar), on the other hand, varying chunk size induces the tipping point speculated above. Performance is also better on the subvolume pattern, which has a higher data-requested to data-in-touched-PGs ratio, reducing the amount of unnecessary data read.

4. Between plane X and Y access patterns, which read roughly the same amount of data, the opaque transforms perform similarly, whereas the transparent transforms perform far better on plane X, which is contiguous relative to the data linearization.

5. ISOBAR outperforms zlib due to superior compression ratio and decompression throughput on the hard-to-compress S3D data.

6. Partial-precision APLOD reads *outperform* none/identity, since the reduced precision enables fewer bytes to be read from storage, whereas full-precision APLOD reads perform slightly slower than none/identity, as they are not completely transparent, and the APLOD data reconstitution process requires some computation.

5.6 Conclusion

To bridge the gap between innovative data transformation methods and real-world adoption, we present a framework for in situ data transformation in scientific applications through the ADIOS I/O middleware. The framework supports data transformation services in a user-transparent and runtime-configurable manner, has decoupled I/O transport and data transformation services, and realizes read-optimizing transforms through a flexible read model.

We evaluate the data transform framework with the large-scale QLG scientific simulation at up to 8,192 cores, as well as through parallel read tests, with no significant overhead. We also explore the general performance implications of data transformations with respect to chunk size, access pattern, and transform "read opacity".

Chapter 6

Conclusion

6.1 Conclusion

For almost three decades, the bitmap index has been used in database systems to accelerate query workloads. As an alternative to other traditional options, such as the B+ tree and hash indexes, bitmap indexes are well suited for data warehouse applications, and more recently, for scientific data exploration. Being recognized as its own class of index, the bitmap index has been the subject of significant research to develop new extensions and variants, such as techniques for compression, binning, and encoding.

However, we perceive recent development the related yet markedly distinct ALACRITY index as a challenge to the monolithic nature of the bitmap index, raising two key questions: first, "Do other, related-yet-distinct indexing methods also exist?"; and second, "Can we generalize the bitmap, ALACRITY, and any other related indexes under a single, more general conceptual model encompassing all?"

In this work, we present the Hyperdyadic Tree index as a constructive affirmation of the former question. Additionally, we demonstrate promising practical characteristics of this index, such as substantially-higher index compression versus the commonly-used WAH-compressed bitmap index without compromise to query performance. This result is confirmed by both empirical measurement as well as theoretical analysis.

We then delve deeper and explicate the commonalities between the bitmap, ALACRITY, and HD-tree indexes with an abstract formal model of the generalized "value-sliced index." Our model accommodates diverse prior work in bitmap compression, binning, and encoding methodologies, as well as the more recent work with ALACRITY and now HD-tree indexes. Besides being of academic interest, this model also forms the theoretical basis of a concrete indexing and query system, PIQUE,

capable of coordinating and delivering a wide range of existing techniques within a single software framework.

Finally, we also address issues limiting the real-world adoption of indexing and other "data transformations" in an *in situ* context within scientific workflows. As in situ and in transit processing become more important with increasing data scale, our work building data transform services within I/O middleware enables clean integration with existing data pipelines.

6.2 Future Work

As a significant portion of this work is concerned with laying theoretical foundations in indexing and query, it is our hope that it will also have impact beyond our main contributions by facilitating further work in this area. Pursuant to this, we conclude with a collection of thoughts on possible avenues for future work.

6.2.1 Combining the HD-tree Index with Index Encodings

Another intriguing facet of the generalized indexing model, and by extension PIQUE, is the potential to combine the new HD-tree RSet representation with existing index encodings (originally intended only for bitmap indexes) "out of the box," so to speak. Exploring the performance implications of this strategy would be an interesting study, and would give some guidance on how best to use the HD-tree RSet representation.

We anticipate the trend of substantially smaller indexes vs. WAH to continue, which is notable since most encodings (substantially) increase index size versus equality encoding when compression is in use. However, the efficiency of our bulk set operation algorithm for HD-trees is reduced relative to WAH bitmaps for the low-arity set operations used in non-equality encodings (e.g., one binary set difference operation under interval encoding versus a high-arity union of bins for equality encoding to answer the same range query). Therefore, improving the efficiency of the original binary set operation algorithm from the CBLQ paper would be necessary to maintain high compute performance for HD-trees under non-equality encodings.

6.2.2 Other RSet Representations, Quantizations, and Encodings in PIQUE

Our model's abstraction of RSet representation for indexing opens up many possibilities for future work. Besides our adaptation of CBLQ quadtree coding to create the HD-tree, several other quadtree-based codings are candidates for such adaptation, including the Fixed Binary Linear Quadtree (or FBLQ) [20] and earlier Linear Quadtree (LQT) [30] and DF-expressions [36, 44]. The FBLQ closely

related to the CBLQ as a *breadth-first* structure, and as a *dense* coding in that all RIDs are explicitly marked as either present or absent; thus, we conjecture the FBLQ would behave similarly to our HD-tree if properly adapted to the indexing context. In contrast, LQTs and DF-expressions are *depth-first* and *sparse* in nature, and might exhibit very different performance characteristics.

Another area of potential interest is the application of *lossy* compression techniques to value-sliced indexing. Previous Bloom filter-based work [6] studied this idea for bitmap indexes, but with the new context of RSet representations, it may be possible to leverage more general lossy (image) compression techniques as well. Though our indexing model likely cannot handle the concept of lossy RSet representations as-is (particularly on the query side), we believe it represents a viable starting point for an extended model that can.

6.2.3 Advanced Parallel Indexing in the PIQUE Platform

PIQUE makes use of a horizontal data partitioning approach to parallelizing the indexing process, as is common in previous systems including FastQuery [23]. The advantage to this approach is its embarrassingly-parallel nature, which yields strong scalability. However, partitioning the data too finely can have a significant negative impact on query performance, as explored in work on the DIRAQ system (parallel ALACRITY) [48]. Such fine partition may be driven either by a desire for extreme indexing parallelism, or by *in situ indexing* where each core performing indexing can only operate on its local data, which may already be finely divided.

The DIRAQ system demonstrates a novel approach to partial index aggregation, in which groups of cores collaborate by merging highly-fragmented, locally-built indexes into a larger, less-fragmented index. The DIRAQ approach is notable in that it leverages hardware-based Remote Memory Access to perform high-throughput *in network* index aggregation. The aggregation strategy is based on certain properties of the ALACRITY index's inverted lists, in particular the possibility to easily concatenate these structures without additional processing.

Having brought ALACRITY under the umbrella of a more general indexing model, we believe it is worth revisiting the DIRAQ method. In particular, it is conceivable the properties of ALACRITY that enable this specialized index aggregation might be shared by other RSet representations, current or future. If these properties could be formalized, they may form the basis for a "DIRAQ extension" to our indexing model, which could in turn enable generalized DIRAQ-like indexing in PIQUE.

Other parallel indexing techniques, such as the hybrid MPI+X parallelism demonstrated on the FastQuery platform [14], would also be worth evaluating in this generalized index context for the purpose of extending PIQUE.

6.2.4 Trans-RSet representation Query Processing

Recent work in optimizing ALACRITY query performance has developed a novel technique for performing set operations between PFOR-Delta-compressed inverted lists via on-the-fly conversion to (uncompressed) bitmaps [104]. Instead of decompressing two lists and then performing a list merge, or even decompressing then converting to bitmap, this method directly decompresses PFOR-Delta chunks into a bitmap without materializing an intermediate inverted list. Furthermore, this approach demonstrates how to perform a bitmap intersection during this decompression, i.e., how to decompress into an existing bitmap, performing a logical AND between the existing bitmap and the contents of the compressed inverted list. This strategy is shown to significantly improve the performance of multi-variable query processing in the ALACRITY PFOR-Delta inverted index.

In the same vein as applying DIRAQ concepts to PIQUE, we believe there is potential in generalizing this on-the-fly bitmap conversion query processing approach beyond ALACRITY. We describe this concept as an *in-place RSet representation converter*. While an *out-of-place RSet representation converter* simply converts a RSet from one RSet representation to another (e.g., decompressing a WAH bitmap to a regular bitmap), an in-place converter is capable of combining a RSet of one RSet representation into a second RSet of a different RSet representation via some set operation (union, intersection, etc.) and without an intermediate result (e.g., decompressing a PFOR-Delta inverted list into an existing bitmap). Under our conceptual model, such a converter could be defined as a function $C : \mathcal{R}_1 \times \mathcal{R}_2 \times \{\cup, \cap, \setminus, \text{ etc.}\} \rightarrow \mathcal{R}_2$, with the flexibility that, from an implementation standpoint, the contents of the second input RSet may be clobbered in the process.

Because in-place RSet representation conversion performs set operations during the conversion, it is a viable alternate approach to fulfilling the `evaluate`(X) step of the generalized query algorithm in Section 3.3.2. That is, instead of performing set operations between encoded RSets directly, this approach would in-place convert all RSets into a common output RSet of a different kind, following the set operations specified by the decoding expression. Thus, in-place conversion is compatible with PIQUE's architecture.

This technique has two immediate implications. First, for some RSet representations like compressed inverted lists, it may substantially improve query performance to use in-place conversion in place of direct set operations, as it does for PFOR-Delta lists. But second, and perhaps more interestingly, this approach can enable query processing over an index that utilizes *more than one RSet representation* within the same structure. This could enable an index to employ HD-trees to store half of its encoded RSets and compressed bitmaps to store the other, for instance. Both compressed bitmaps and HD-trees could then be combined to answer queries via on-the-fly conversion into a shared uncompressed bitmap by employing the appropriate conversion algorithms for each RSet

representation.

This flexibility would enable finely-tuned selection of RSet representations for each encoded RSet in the index, rather than being restricted to an all-or-nothing decision between a bitmap index, HD-tree index, ALACRITY index, etc. Besides potentially allowing for even higher index compression than is currently possible, is is conceivable that other benefits might be found, such as interesting cross-RSet representation query performance implications.

We have done some preliminary work on in-place converters in PIQUE already, having developed both a generalized in-place conversion based query algorithm and a highly-optimized in-place converter for HD-tree-to-bitmap conversion, and have obtained some promising preliminary results. We believe that continuing this experimentation to formally evaluate the performance impact on-the-fly conversion query processing, as well as developing similar algorithms for other RSet representations, is a very promising direction for future exploration.

6.2.5 Extensions to the Generalized Indexing and Query Model

In developing the abstract indexing and query model presented Section 3.3, we have noted some areas for potential extension. In particular, the model as stated does not address the notion of *candidate checks* during query processing.

We believe it should be possible to incorporate candidate checking into both the abstract model and PIQUE by building on already-defined concepts. We envision one possible approach to candidate checking as follows:

1. Use the index decoder algorithm to recover *hit* and *edge* bin RSets separately, rather than as a single RSet as is done now.
2. Convert the edge bin RSets to RIDs, suitable for probing the original data.
3. Convert the set of confirmed candidate RIDs back into an RSet.
4. Combine the confirmed candidate RSet with the hit RSet to form a final result RSet.
5. Continue with the query processing algorithm as before, converting the result RSet to RIDs for the user.

Whether this general approach precludes existing candidate check optimizations for bitmap indexing, how it meshes with multi-variate queries, and whether other generalized strategies may exist, are questions for further investigation.

6.2.6 Indexing, Querying, and Other Data Services in I/O Middleware

While the ADIOS data transforms framework successfully enables in situ data transformations at the I/O middleware level, including indexing, we see three directions for future work here.

First, though indexing can be effected as a data transformation, it is perhaps more natural to consider it a *data derivative*, defined as a process for deriving a new artifact from existing data without modifying that data. We see data derivatives as the third type of *data service*, alongside data transformations (explored in Chapter 5) and I/O transports (explored by ADIOS itself). Other in situ data derivatives under active research include situ analytics [1, 103], feature extraction [101], visualization [55, 100] and non-clustered indexing [45]. Seamless data derivative services in I/O middleware, in the same manner as data transforms are treated in this paper, could similarly aid the widespread adoption of these innovative technologies.

Second, based on our experience here, it has become apparent that some complex methods span beyond this basic trichotomy of data service classes. For instance, the DIRAQ indexing workflow [48] incorporates aspects of both clustered indexing/compression (data transformation) and in-network index merging and I/O aggregation (I/O transport). Similarly, ISOBAR hybrid compression [73] asynchronously overlaps compression (data transformation) with I/O transport of incompressible data. In both cases, the data transform and I/O transport components cannot be distilled as sequential phases, but are inextricably linked. This cross-cutting nature makes fitting them into a single framework difficult; however, if achieved, the integration of such complex data pipeline techniques in the I/O middleware layer could have significant impact across entire scientific workflows.

Third, while in situ indexing within I/O middleware is useful alone to achieve data pipeline compatibility, the development of a general-purpose query layer within I/O middleware would compound this advantage. Some of the authors have already participated in the initial development of such a layer for ADIOS [12], and both ALACRITY and FastBit have been integrated; a next logical step would be to integrate PIQUE, as well.

BIBLIOGRAPHY

[1] Abbasi, H. et al. “DataStager: scalable data staging services for petascale applications”. English. *Cluster computing* **13**.3 (2010), pp. 277–290.

[2] Abbasi, H. et al. “Just in time: adding value to the IO pipelines of high performance applications with JITStaging”. *Proceedings of the ACM international symposium on high-performance parallel and distributed computing (HPDC)*. 2011, pp. 27–36.

[3] Adams, M. & Ward, R. “Wavelet transforms in the JPEG-2000 standard”. *Proceedings of the ieee pacific rim conference on communications, computers and signal processing (PACRIM)*. Vol. 1. 2001, 160–163 vol.1.

[4] Antoshenkov, G. “Byte-aligned bitmap compression”. *Proceedings of the data compression conference (DCC)*. 1995, pp. 476–.

[5] Antoshenkov, G. & Ziauddin, M. “Query processing and optimization in oracle rdb”. English. *VLDB journal* **5**.4 (1996), pp. 229–237.

[6] Apaydin, T. et al. “Approximate encoding for direct access and query processing over compressed bitmaps”. *Proceedings of the international conference on very large data bases (VLDB)*. VLDB ’06. Seoul, Korea: VLDB Endowment, 2006, pp. 846–857.

[7] Bell, S. et al. “Spatially referenced methods of processing raster and vector data”. *Image and vision computing* **1**.4 (1983), pp. 211–220.

[8] Bernardo, G. et al. “Compact querieable representations of raster data”. *String processing and information retrieval (SPIR)*. Vol. 8214. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 96–108.

[9] Bowers, K. J. et al. “0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner”. *Proceedings of the ACM/IEEE conference on supercomputing*. SC ’08. Austin, Texas: IEEE Press, 2008, pp. 1–11.

[10] Bowers, K. J. et al. “Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulationa)”. *Physics of plasmas* **15**.5, 055703 (2008).

[11] Bowers, K. J. et al. “Advances in petascale kinetic plasma simulation with VPIC and roadrunner”. *Journal of physics: conference series* **180**.1 (2009), p. 012055.

[12] Boyuka II, D. A. et al. “The hyperdyadic index and generalized indexing and query with PIQUE”. *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. SSDBM ’15. La Jolla, California: ACM, 2015, pp. 1–12.

[13] Burtscher, M. & Ratanaworabhan, P. “FPC: a high-speed compressor for double-precision floating-point data”. *IEEE transactions on computers* **58**.1 (2009), pp. 18–31.

[14] Byna, S. et al. “Parallel I/O, analysis, and visualization of a trillion particle simulation”. *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. SC ’12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, pp. 1–12.

[15] Chambi, S. et al. “Better bitmap performance with roaring bitmaps”. *Software: practice and experience* (2015), n/a–n/a.

[16] Chan, C.-Y. & Ioannidis, Y. E. “Bitmap index design and evaluation”. *Proceedings of the ACM international conference on management of data (SIGMOD)*. SIGMOD ’98. Seattle, Washington, USA: ACM, 1998, pp. 355–366.

[17] Chan, C.-Y. & Ioannidis, Y. E. “An efficient bitmap encoding scheme for selection queries”. *SIGMOD record* **28**.2 (1999), pp. 215–226.

[18] Chang, C.-C. et al. “Code transformation of DF-expression between bintree and quadtree”. *Proceedings of the joint conference of the international conference on information, communications and signal processing (ICICS) and IEEE pacific-rim conference on multimedia (PCM)*. Vol. 1. 2003, 559–563 Vol.1.

[19] Chang, C.-C. et al. “Code transformation algorithms for two breadth-first linear quadtrees”. *Proceedings of the education technology and training (ETT) and geoscience and remote sensing (GRS) workshop*. Vol. 1. 2008, pp. 799–802.

[20] Chang, H. K. & Chang, J.-W. *Fixed binary linear quadtree coding scheme for spatial data*. 1994.

[21] Chen, P.-M. “Variant code transformations for linear quadtrees”. *Pattern recognition letters* **23**.11 (2002), pp. 1253–1262.

[22] Chen, Z. et al. “A survey of bitmap index compression algorithms for big data”. *Tsinghua science and technology* **20**.1 (2015), pp. 100–115.

[23] Chou, J. et al. “Parallel index and query for large scale data analysis”. *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. SC ’11. Seattle, Washington: ACM, 2011, pp. 1–11.

[24] Comer, D. “Ubiquitous b-tree”. *ACM computing surveys* **11**.2 (1979), pp. 121–137.

[25] Deri, L. et al. “Collection and exploration of large data monitoring sets using bitmap databases”. English. *Proceedings of the international workshop on traffic monitoring and analysis (TMA)*. Vol. 6003. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 73–86.

[26] Dong, B. et al. “Heavy-tailed distribution of parallel I/O system response time”. *Proceedings of the workshop on petascale data storage (PDSW)*. PDSW ’15. Austin, Texas: ACM, 2015, pp. 37–42.

[27] Dorier, M. et al. “Damaris: how to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O”. *Proceedings of the IEEE international conference on cluster computing (CLUSTER)*. 2012, pp. 155–163.

[28] Folk, M. et al. “An overview of the HDF5 technology suite and its applications”. *Proceedings of the joint EDBT/ICDT conference workshop on array databases*. AD ’11. Uppsala, Sweden: ACM, 2011, pp. 36–47.

[29] Franklin, M. et al. *ACM SIGMOD record* **34**.4 (2005), pp. 27–33.

[30] Gargantini, I. “An effective way to represent quadtrees”. *Communcations of the ACM* **25**.12 (1982), pp. 905–910.

[31] Gong, Z. et al. “MLOC: multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns”. *Proceedings of the IEEE international conference on parallel processing (ICPP)*. 2012, pp. 239–248.

[32] Gong, Z. et al. “PARLO: PArallel run-time layout optimization for scientific data explorations with heterogeneous access patterns”. *Proceedings of the ACM/IEEE international symposium on cluster, cloud and grid computing (CCGrid)*. 2013, pp. 343–351.

[33] Gosink, L. et al. “HDF5-FastQuery: accelerating complex queries on HDF datasets using fast bitmap indices”. *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. 2006, pp. 149–158.

[34] Goyal, N. & Sharma, Y. “New binning strategy for bitmap indices on high cardinality attributes”. *Proceedings of the bangalore annual compute conference*. COMPUTE ’09. Bangalore, India: ACM, 2009, pp. 1–5.

[35] Hellerstein, J. M. et al. “Generalized search trees for database systems”. *Proceedings of the international conference on very large data bases (VLDB)*. VLDB ’95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 562–573.

[36] Huang, C.-Y. & Chung, K.-L. “Transformations between bincodes and the DF-expression”. *Computers & graphics* **19**.4 (1995). Computer Graphics Art, pp. 601–610.

[37] Hunter, A. & Willis, P. “Classification of quad-encoding techniques”. *Computer graphics forum* **10**.2 (1991), pp. 97–112.

[38] Hunter, A. & Willis, P. J. "Breadth-first quad encoding for networked picture browsing". *Computers & graphics* **13**.4 (1989), pp. 419–432.

[39] *IEEE standard for binary floating-point arithmetic*. IEEE. 2008.

[40] Jenkins, J. et al. "Byte-precision level of detail processing for variable precision analytics". *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. 2012, pp. 1–11.

[41] Jenkins, J. et al. "Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying". English. *Proceedings of the international conference on database and expert systems applications (dexa)*. Vol. 7447. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 16–30.

[42] Jenkins, J. et al. "ALACRITY: analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying". English. *Transactions on large-scale data- and knowledge-centered systems x (TLDKS X)*. Vol. 8220. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 95–114.

[43] Johnson, T. "Performance measurements of compressed bitmap indices". *Proceedings of the international conference on very large data bases (VLDB)*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–289.

[44] Kawaguchi, E. & Endo, T. "On a method of binary-picture representation and its application to data compression". *IEEE transactions on pattern analysis and machine intelligence (TPAMI)* **PAMI-2**.1 (1980), pp. 27–35.

[45] Kim, J. et al. "Parallel in situ indexing for data-intensive computing". *Proceedings of the IEEE symposium on large data analysis and visualization (LDAV)*. 2011, pp. 65–72.

[46] Kleinrock, L. *Queueing systems*. Vol. 1. John Wiley & Sons, Inc., 1975.

[47] Lakshminarasimhan, S. et al. "ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data". *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. SC '11. Seattle, Washington: ACM, 2011, pp. 1–11.

[48] Lakshminarasimhan, S. et al. "Scalable in situ scientific data encoding for analytical query processing". *Proceedings of the ACM international symposium on high-performance parallel and distributed computing (HPDC)*. HPDC '13. New York, New York, USA: ACM, 2013, pp. 1–12.

[49] Li, J. et al. "Parallel netCDF: a high-performance scientific I/O interface". *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. 2003, p. 39.

[50] Lin, T.-W. "Compressed quadtree representations for storing similar images". *Image and vision computing* **15**.11 (1997), pp. 833–843.

[51] Lin, T.-W. "Set operations on constant bit-length linear quadtrees". *Pattern recognition* **30**.7 (1997), pp. 1239–1249.

[52] Lindstrom, P. & Isenburg, M. "Fast and efficient compression of floating-point data". *IEEE transactions on visualization and computer graphics (TVCG)* **12**.5 (2006), pp. 1245–1250.

[53] Lofstead, J. et al. "Six degrees of scientific data: reading patterns for extreme scale science IO". *Proceedings of the ACM international symposium on high-performance parallel and distributed computing (HPDC)*. HPDC '11. San Jose, California, USA: ACM, 2011, pp. 49–60.

[54] Lofstead, J. F. et al. "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)". *Proceedings of the international workshop on challenges of large applications in distributed environments (CLADE)*. CLADE '08. Boston, MA, USA: ACM, 2008, pp. 15–24.

[55] Ma, K.-L. "In situ visualization at extreme scale: challenges and opportunities". *IEEE computer graphics and applications* **29**.6 (2009), pp. 14–19.

[56] National Energy Research Scientific Computing Center (NERSC). *Edison*. 2015. URL: `https://www.nersc.gov/users/computational-systems/edison/` (visited on 12/23/2015).

[57] National Energy Research Scientific Computing Center (NERSC). *File storage and I/O: edison file systems*. 2015. URL: `https://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/` (visited on 12/23/2015).

[58] Oak Ridge National Lab. *ADIOS*. 2015. URL: `https://www.olcf.ornl.gov/center-projects/adios/` (visited on 12/23/2015).

[59] Oak Ridge National Lab. *ADIOS on GitHub*. 2015. URL: `https://github.com/ornladios/ADIOS` (visited on 12/23/2015).

[60] O'Neil, P. & Quass, D. "Improved query performance with variant indexes". *Proceedings of the ACM international conference on management of data (sigmod)*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 38–49.

[61] O'Neil, P. E. "MODEL 204 architecture and performance". English. *Proceedings of the international workshop on high performance transaction systems*. Vol. 359. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, pp. 39–59.

[62] Pascucci, V. & Frank, R. "Global static indexing for real-time exploration of very large regular grids". *Proceedings of the ACM/IEEE conference on supercomputing*. 2001, pp. 45–45.

[63] Polte, M. et al. "...and eat it too: high read performance in write-optimized HPC I/O middleware file formats". *Proceedings of the workshop on petascale data storage (PDSW)*. PDSW '09. Portland, Oregon: ACM, 2009, pp. 21–25.

[64] Roelofs, G. et al. *Zlib*. 2015. URL: `http://www.zlib.net/` (visited on 12/23/2015).

[65] Rotem, D. et al. "Minimizing I/O costs of multi-dimensional queries with bitmap indices". *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. 2006, pp. 33–44.

[66] Rotem, D. et al. "Optimizing candidate check costs for bitmap indices". *Proceedings of the ACM international conference on information and knowledge management (CIKM)*. CIKM '05. Bremen, Germany: ACM, 2005, pp. 648–655.

[67] Rotem, D. et al. "Optimizing I/O costs of multi-dimensional queries using bitmap indices". English. *International conference on database and expert systems applications (DEXA)*. Vol. 3588. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 220–229.

[68] Rübel, O. et al. "High performance multivariate visual data exploration for extremely large data". *Proceedings of the ACM/IEEE international conference on high performance computing, networking, storage and analysis (supercomputing)*. SC '08. Austin, Texas: IEEE Press, 2008, pp. 1–12.

[69] Samatova, N. F. *Big data analytics group*. 2015. URL: `http://www.freescience.org/cs/` (visited on 12/23/2015).

[70] Samet, H. "The quadtree and related hierarchical data structures". *ACM computng surveys (CSUR)* **16**.2 (1984), pp. 187–260.

[71] Samet, H. "Data structures for quadtree approximation and compression". *Communications of the ACM* **28**.9 (1985), pp. 973–993.

[72] Samet, H. *Applications of spatial data structures: computer graphics, image processing, and GIS*. Addison-Wesley, 1990.

[73] Schendel, E. R. et al. "ISOBAR hybrid compression-i/o interleaving for large-scale parallel I/O optimization". *Proceedings of the ACM international symposium on high-performance*

parallel and distributed computing (HPDC). HPDC '12. Delft, The Netherlands: ACM, 2012, pp. 61–72.

[74] Seward, J. *Bzip2*. 2015. URL: `http://www.bzip.org/` (visited on 12/23/2015).

[75] Shoshani, A. et al. "SDM center technologies for accelerating scientific discoveries". *Journal of physics: conference series* **78**.1 (2007), p. 012068.

[76] Sinha, R. R. & Winslett, M. "Multi-resolution bitmap indexes for scientific data". *ACM transactions on database systems (TODS)* **32**.3 (2007).

[77] Stockinger, K. "Design and implementation of bitmap indices for scientific data". *International conference on database and expert systems applications (DEXA)*. 2001, pp. 47–57.

[78] Stockinger, K. et al. "Query-driven visualization of large data sets". *Proceedings of the IEEE conference on visualization (VIS)*. 2005, pp. 167–174.

[79] Stockinger, K. & Wu, K. "Bitmap indices for data warehouses". *Data warehouses and OLAP: concepts, architectures, and solutions* (2006), p. 57.

[80] Stockinger, K. & Wu, K. "Bitmap indices for data warehouses". *Data warehouses and OLAP: concepts, architectures and solutions*. IRM Press, 2007, pp. 157–178.

[81] Stockinger, K. et al. "Improving the performance of high-energy physics analysis through bitmap indices". English. *International conference on database and expert systems applications (DEXA)*. Vol. 1873. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 835–845.

[82] Stockinger, K. et al. "Evaluation strategies for bitmap indices with binning". English. *International conference on database and expert systems applications (DEXA)*. Vol. 3180. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 120–129.

[83] Vahala, G. et al. "Poincar'e recurrence and spectral cascades in three-dimensional quantum turbulence". *Physical review e* **84**.4 (2011), p. 046713.

[84] Vahala, G. et al. "Unitary qubit lattice simulations of multiscale phenomena in quantum turbulence". *Proceedings of the international conference on high performance computing, networking, storage and analysis (supercomputing)*. SC '11. Seattle, Washington: ACM, 2011, pp. 1–11.

[85] Wong, H. K. T. et al. "Bit transposed files". *Proceedings of the international conference on very large data bases (VLDB)*. VLDB '85. Stockholm, Sweden: VLDB Endowment, 1985, pp. 448–457.

[86] Wong, H. et al. “Bit transposition for very large scientific and statistical databases”. English. *Algorithmica* **1**.1-4 (1986), pp. 289–309.

[87] Wu, K et al. “FastBit: interactively searching massive data”. *Journal of physics: conference series* **180**.1 (2009), p. 012053.

[88] Wu, K. “FastBit: an efficient indexing technology for accelerating data-intensive science”. *Journal of physics: conference series* **16**.1 (2005), p. 556.

[89] Wu, K. et al. “A performance comparison of bitmap indexes”. *Proceedings of the ACM international conference on information and knowledge management (CIKM)*. CIKM ’01. Atlanta, Georgia, USA: ACM, 2001, pp. 559–561.

[90] Wu, K. et al. “Compressing bitmap indexes for faster search operations”. *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. 2002, pp. 99–108.

[91] Wu, K. et al. “Using bitmap index for interactive exploration of large datasets”. *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. 2003, pp. 65–74.

[92] Wu, K. et al. “On the performance of bitmap indices for high cardinality attributes”. *Proceedings of the international conference on very large data bases (VLDB)*. VLDB ’04. Toronto, Canada: VLDB Endowment, 2004, pp. 24–35.

[93] Wu, K. et al. “Optimizing bitmap indices with efficient compression”. *ACM transactions on database systems (TODS)* **31**.1 (2006), pp. 1–38.

[94] Wu, K. et al. *Performance of multi-level and multi-component compressed bitmap indices*. Tech. rep. 60891. Lawrence Berkeley National Lab, 2007.

[95] Wu, K. et al. “Analyses of multi-level and multi-component compressed bitmap indexes”. *ACM transactions on database systems (TODS)* **35**.1 (2008), pp. 1–52.

[96] Wu, K. et al. “Breaking the curse of cardinality on bitmap indexes”. English. *Proceedings of the international conference on scientific and statistical database management (SSDBM)*. Vol. 5069. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 348–365.

[97] Wu, K. et al. “Finding regions of interest on toroidal meshes”. *Computational science & discovery* **4**.1 (2011), p. 015003.

[98] Wu, M.-C. & Buchmann, A. “Encoded bitmap indexing for data warehouses”. *Proceedings of the international conference on data engineering (ICDE)*. 1998, pp. 220–230.

[99] Yeh, P.-S. et al. “Implementation of CCSDS lossless data compression in HDF”. *Proceedings of the earth science technology conference*. 2002.

[100] Yu, H. et al. “In situ visualization for large-scale combustion simulations”. *IEEE computer graphics and applications* **30**.3 (2010), pp. 45–57.

[101] Zhang, F. et al. “In-situ feature-based objects tracking for large-scale scientific simulations”. *Sc companion: proceedings of the international conference on high performance computing, networking, storage and analysis (SCC)*. 2012, pp. 736–740.

[102] Zhang, Z. et al. “Scientific computing meets big data technology: an astronomy use case”. *Arxiv preprint arxiv:1507.03325* (2015).

[103] Zheng, F. et al. “PreDatA - preparatory data analytics on peta-scale machines”. *Proceedings of the IEEE international parallel and distributed processing symposium (IPDPS)*. 2010, pp. 1–12.

[104] Zou, X. et al. “Fast set intersection through run-time bitmap construction over PForDelta-compressed indexes”. English. *Proceedings of the Euro-Par conference on parallel processing*. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 668–679.

[105] Zukowski, M. et al. “Super-scalar RAM-CPU cache compression”. *Proceedings of the international conference on data engineering (ICDE)*. 2006, pp. 59–59.