THE UNIVERSITY OF CHICAGO

EXPLOITING DOMAIN-SPECIFIC DATA PROPERTIES TO IMPROVE COMPRESSION FOR HIGH ENERGY PHYSICS DATA

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ARJUN RAWAL

CHICAGO, ILLINOIS
JUNE 2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my appreciation for the many friends and colleagues that enabled this project and inspired my work. I would also like to extend my sincerest gratitude to Professor Andrew A. Chien for his guidance, support, and inspiration throughout my time at the University of Chicago. I would also like to thank Professor Raul Castro Fernandez and Professor Robert Gardner for their valuable advice, and for taking the time to serve on my thesis committee.

I would also like to thank the current and former members of the LSSG for their mentorship and friendship throughout the course of my research. Finally, I want to thank my family and friends for their continuous encouragement, support, and love throughout the process of writing this thesis.

ABSTRACT

Data storage is a fundamental concern for high energy physics; the experiments and data analysis needed to discover new results require petabytes of measurements from particle collisions. Accordingly, data compression has been a central focus of data storage solutions, as it provides an effective way to reduce storage costs and improve analysis performance. Whereas interactive analysis workloads benefit from fast data availability for computation, archival storage benefits from compression which makes data as small as possible. For most high energy physics data, the standard approach to compression is "one size fits all" — data is stored for archive with the same compression used for interactive analysis. Because data analysis and long term storage are fundamentally different use cases, the tradeoffs made to provide performant data analysis result in relatively poor compression for long term data storage. We propose that high energy physics data could be stored much more compactly if we use modern computational algorithms and compression approaches that take into account the fundamental characteristics of the data.

We study several modern compression algorithms and evaluate their performance on high energy physics data. We then evaluate a variety of techniques used in data compression to improve compression ratio: delta encoding, floating point representation, data aggregation, and dictionary optimizations. These algorithms and techniques exist in a tradeoff space where compression ratio, throughput, and resource utilization can be exchanged to find the best compression for a specific use case.

Evaluation on real datasets from the ATLAS and CMS experiments shows that adopting algorithms designed for modern processors and larger memory sizes can provide compression ratio improvements of 7% while providing better compression and decompression throughput. Furthermore, applying techniques that take into account the underlying type of a block of data, not just the bytes of data, can increase compression ratio by an additional 5%. Overall, we find that an approach that prioritizes compression ratio can reduce the overall size of data files by more than 15%, providing a significant reduction in data storage requirements.

However, this solution is useful only if it is cost-effective. We analyze the cost of scaling up our compression strategies for the ATLAS experiment. We find that a production implementation of our approach would require fewer than 50 CPU cores to handle reading a petabyte of data per day. This approach could reduce data storage requirements by more than 8 petabytes, and save hundreds of thousands of dollars in hard drive and tape storage costs each year. Hence, our approach is cost effective and feasible on a large scale.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Our understanding of the subatomic world is derived largely from particle collider experiments, which create high kinetic energy collisions of projectiles and analyze the byproducts of their impact. These facilities, such as the Large Hadron Collider, contain detectors with millions of electronic output channels, generating incredible quantities of data. Through a combination of custom hardware and software filtering, this data is pared down from many petabytes per second to terabytes per second [4, 18]. Then, this data can be analyzed further at accelerator datacenters before being filtered and analyzed by physicists to determine the fundamental interactions between particles. Even after the extensive filtering and data selection process, petabytes of data are generated every year, and must be retained in full quality to enable further analysis and simulation.

CERN, the European Organization for Nuclear Research, manages these datasets, storing them in datacenters across the globe to enable fast and reliable data access. This massive quantity of data is growing faster than ever before, as particle accelerators collect more and more data. The Large Hadron Collider undergoes a process of Runs and Shutdowns, where data is collected, and sensors and instruments are upgraded, respectively. Each upgrade increases the luminosity of the accelerator, a measure of the accelerator's performance, and increases the quantity of sensor data generated by the instrument.

Data storage for high energy physics must meet a variety of demands. First, data must be stored reliably because it is extremely costly to generate collision data. Secondly, this data must be accessible around the globe for analysis, ideally with low latency access. Finally, the data must be stored compactly, that is, it should take up as little room as possible in data centers to reduce data storage costs. As Figure 1.1 demonstrates, the data is stored in different formats at the Tier 0, Tier 1, and Tier 2 sites [54]. RAW event data, the primary archive of accelerator events, are not used for direct analysis, but are stored at Tier 0 and Tier 1 sites to reconstruct particle interactions. RECO (reconstructed data) are large processed files which contain data about the interactions of the subatomic particles and their hits. AOD (Analysis Object Data) files are a distilled version of the RECO files and mainly contain the reconstructed objects and data about their motion. Petabytes of each type of file are stored in the CERN grid on a mix of tapes and hard drives to provide the best balance of cost, reliability, and capacity. Storing this data reliably and compactly is an increasingly important concern as the Large Hadron Collider approaches Run 3, which is predicted to generate ten times as much data as Run 2 [36].

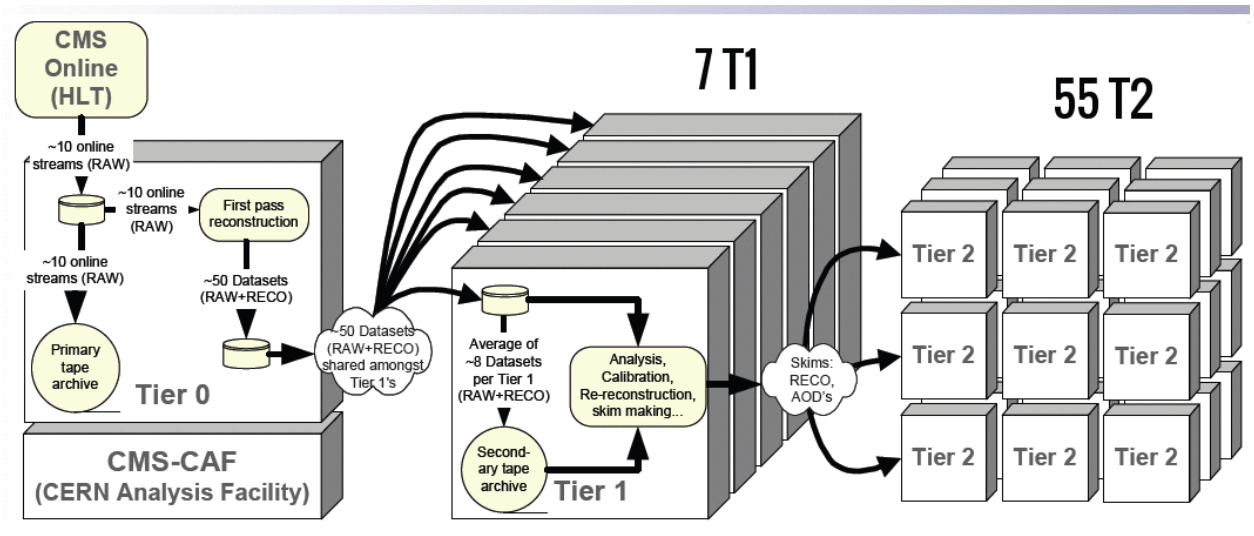

CMS Online (HLT)
~10 online streams (RAW)
~10 online streams (RAW)
First pass reconstruction
~10 online streams (RAW)
~50 Datasets (RAW+RECO)
Primary tape archive
Tier 0
CMS-CAF (CERN Analysis Facility)
~50 Datasets (RAW+RECO) shared amongst Tier 1's
7 T1
Average of ~8 Datasets per Tier 1 (RAW+RECO)
Analysis, Calibration, Re-reconstruction, skim making...
Second-ary tape archive
Tier 1
Skims: RECO, AOD's
55 T2
Tier 2

Figure 1.1: CMS Data Processing Pipeline

The performance of compression algorithms exist in a trade off space, as the algorithms which provide high compression ratio, the ratio of uncompressed to compressed data size, generally have very low throughput, the rate that data can be compressed. On the other hand, the algorithms which can provide high throughput often have much lower compression ratios. Hence, compression performance is not a generally well-defined term, but instead an workload specific balance between throughput and compression ratio.

In recent years, advancements in general purpose data compression have not significantly increased compression ratio. However, major improvements have been made in increasing compression throughput on specific hardware platforms [43, 41, 39]. Additionally, compression for data with known or learned distributions has emerged as one of the best ways to increase compression ratio beyond generalized approaches [46, 20, 22]. These approaches rely on domain-specific insights, that is, predictable features of the data which can be extracted by either human analysis or machine learning techniques. Compression algorithms are increasingly designed with their target data in mind, as general purpose compression performs well across the board, but can be greatly improved by organizing and viewing data in a particular way. Although particle physics data is noisy and utilizes a variety of types, objects, and formats, even the most basic insights about datatypes and byte level formatting can be valuable for designing compression approaches.

Columnar data storage, or storage which aligns the fields of an object or table by column, rather than row, is one of the oldest and most widely used techniques for improving data compression. Columnar storage collocates entries that are most similar to each other, increasing the similarity between neighboring data. For example, a table that stores Event(*date*, *id*, *speed*) tuples will be stored with all dates aligned consecutively. Then, a compression algorithm viewing that block of date fields will be able to recognize the underlying similarity in the bit level date patterns, enabling the algorithm to omit repeated information and save space. Rearranging data for compression, as columnar storage does, is a well studied approach in database theory [5] [33]. ROOT, the primary software used for high energy physics analysis, is based on a columnar storage, and stores object data in a vertical format rather than a horizontal format.

Most high energy physics data is stored in the same format and with the same compression, whether it is being used for backup storage or for interactive analysis. However, data analysis and archival storage are two distinct domains which require fundamentally different

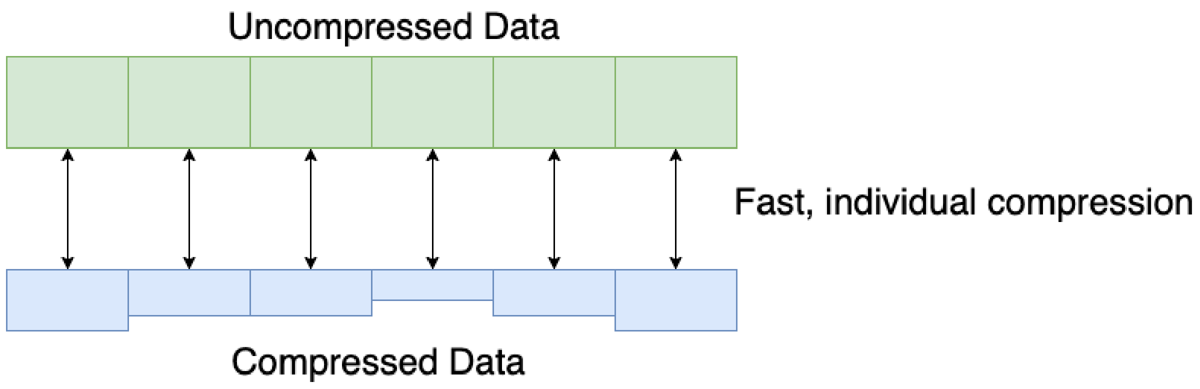


Figure 1.2: Current Compression

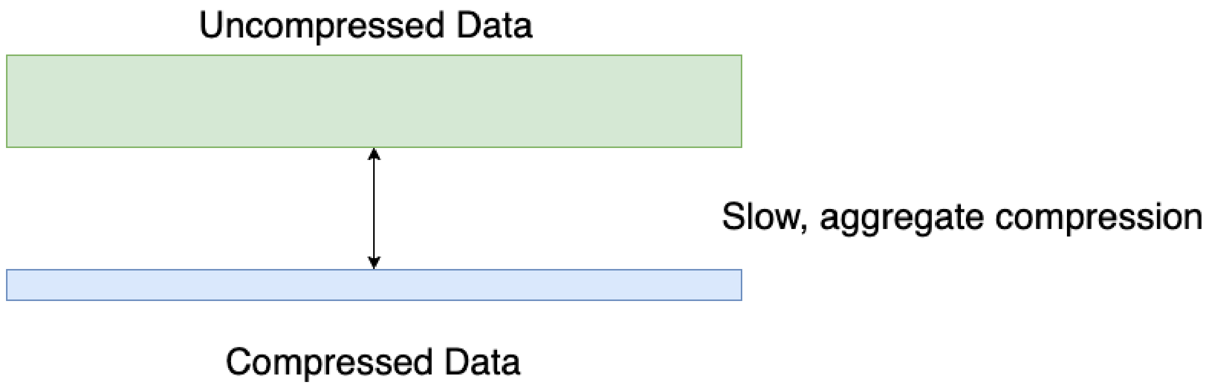


Figure 1.3: Aggregated Data

compression approaches. The high throughput and low latency requirements of data used for analysis is at odds with the need to reduce data storage costs with high compression ratio. This presents a challenge to current data storage techniques, as the standard format is optimized for performance during interactive analysis. More specifically, this data is compressed using algorithms and settings which provide high throughput at the expense of compression ratio, and align data in ways that improve quick reading and writing, but does not always collocate similar data to improve compression ratio (Figure 1.2). Therefore, we see that designing a data compression strategy to meet the goals of long term storage has the capability to greatly improve compression ratio.

We see two main ways in which data compression for high energy physics data storage can be improved. First, data can be aggregated to allow compression to be done in larger

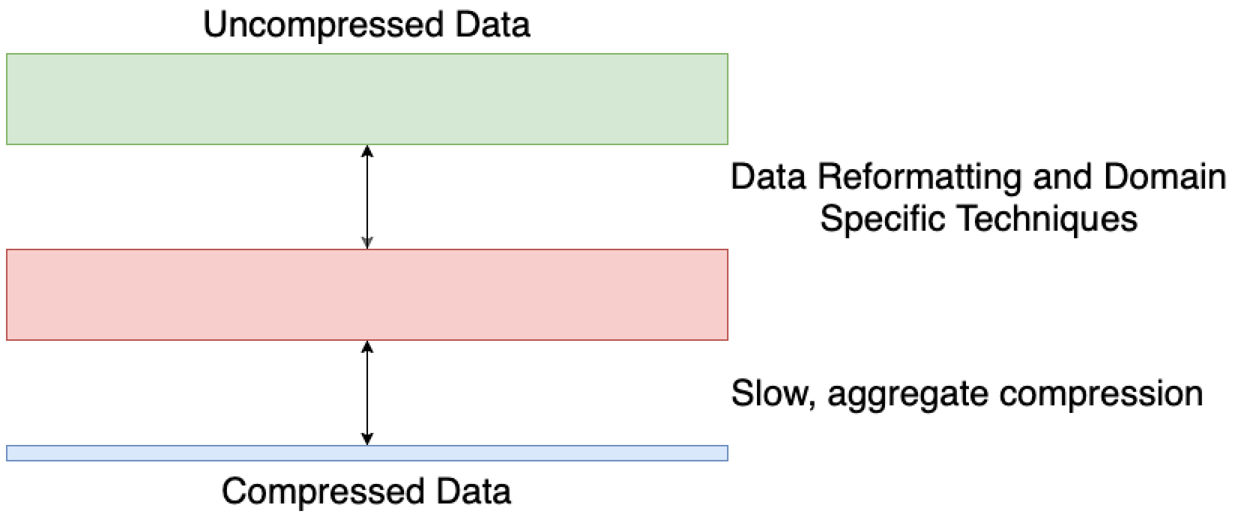

Uncompressed Data
Data Reformatting and Domain Specific Techniques
Slow, aggregate compression
Compressed Data

Figure 1.4: Aggregated Data with Domain Specific Techniques

blocks, increasing compression ratio (Figure 1.3). Second, domain specific information can be used in tandem with compression algorithms to better expose patterns to compression algorithms (Figure 1.4). Domain specific compression and data reformatting has the potential to dramatically increase compression ratio on high energy physics data. However, these compression ratio oriented algorithms and techniques have variable benefit based on the underlying data and can be computationally expensive to implement. Furthermore, storage in a archive-oriented format necessitates additional overhead to recompress or rearrange data. Hence, we must answer the following questions.

1. Do data specific compression techniques improve compression for collision datasets?

2. What compression algorithms, settings, and data layouts give the best compression on these datasets?

3. To what extent are these strategies practical, usable, and cost effective for archival storage?

1.2 Approach

To answer these questions, we begin by studying the state of the art in data compression. We explore the effect that algorithmic configurations and settings have on compression ratio and throughput, and the effectiveness of combining our domain specific techniques with general purpose algorithms. With that understanding, we can then design, implement, and evaluate data specific techniques that use the inherent structure of datatypes, as well as knowledge about data distribution to assist the algorithms in storing data more compactly.

Most data compression algorithms rely on two fundamental insights: previous data patterns are likely to occur again, and repeated bit patterns can be rewritten with a smaller code or reference. We exploit these insights by utilizing transformations and aggregations which represent data in a way that increases repetitions and common bit patterns. Many of these transformations do not actually reduce data size on their own, but when combined with a compression algorithm can increase compression ratio due to a more easily compressed distribution of data.

We propose that storing data in a storage focused format, targeted towards providing better compression ratio, is feasible and would save petabytes of data each year. This format has a few key features which lead to more compact data storage. First, we aggregate data in larger blocks to take advantage of larger system memory and allow compression algorithms to better extract data patterns. This is not possible in an analysis focused format, as the larger block sizes slow down random reads because entire blocks must be decompressed before they can be read. Secondly, we utilize higher levels of compression algorithms. These levels use larger dictionaries and are more resource intensive, but deliver better compression ratio. Finally, we use a set of focused techniques which use our understanding of the underlying data to transform it into a more compressible representation. These techniques have high throughput and are extremely configurable, and therefore can be used in both analysis focused and storage focused compression.

To understand the effectiveness of our approach, we look to two of the largest ongoing experiments at the Large Hadron Collider(LHC): ATLAS (A Toroidal LHC ApparatuS) and CMS (Compact Muon Solenoid). We use CMS and ATLAS datasets to understand the types of data stored in high energy physics files and the performance of current data compression strategies. These representative samples contain thousands of columns and millions of events, with floating point, integer, boolean, and other structured object data.

1.3 Contributions

The results of our compression analysis demonstrate that better data layout choices, aggregate columnar storage, and the use of `zstd`, tuned for large window sizes and memory usage (level$\geq$9), can improve the compression ratio of high energy physics data by more than 15% over current compression approaches. We find that this benefit is achievable with high (10x greater), but cost-effective computational load during compression, and equivalent or better decompression throughput. The specific contributions of this thesis are as follows:

1. We evaluate the performance of 3 of the most used general purpose compression algorithms, Zlib, Snappy, and Zstandard, on high energy physics data, and analyze how their configurations can be tuned to meet performance goals. We find that increasing compression levels have diminishing returns for compression ratio, but simply moving from `zlib`, level=5 to `zstd`, level=7 can reduce overall file size by 4% with a 40% increase in both compression and decompression throughput.

2. We study how techniques such as delta encoding and float splitting, which use knowledge of the underlying data to represent it in a more compressible way, affect compression throughput and compression ratio. We find that delta encoding and float splitting can improve compression ratio by 2.5% and 4.7% on integer and floating point data, respectively, with less than a 2% cost on throughput.

3. We study how data aggregation across files and pretraining dictionaries can affect overall compression ratio, and how to best select data for pretraining and aggregation. We find that these techniques can improve compression throughput by up to 3x for high compression levels, and increase compression ratio by up to 2%.

4. We design and model five different strategies for reducing the storage requirements for the ATLAS experiment. We find that relatively simple configuration changes can save more than 3 petabytes of storage from the current 85 petabytes, and that more complex strategies can save upwards of 10 petabytes, with only a modest increase in data center processing required. We estimate the CPU, petabyte, and dollar costs of the five approaches over the next 15 years using projected data storage requirements from the ATLAS experiment.

1.4 Outline

The rest of this thesis is organized as follows. In Chapter 2 we present background on current approaches to high energy physics data storage and compression techniques. In Chapter 3 we characterize the experimental setup and methodology, and describe the techniques that we evaluate for improved compression on the different types present in the data. Chapter 4 presents our detailed experimental results on 2 different datasets, and provides selected results on the various techniques we chose to evaluate. In Chapter 5, we design production compression strategies using our results from Chapter 4 and estimate their impact on the current and future data storage infrastructure. We discuss related work on data compression and scientific data storage in Chapter 6. Finally, in Chapter 7, we summarize our results and provide a discussion of future areas of exploration.

CHAPTER 2
BACKGROUND

In this chapter, we describe the current storage techniques used for high energy physics data and discuss how compression algorithms are designed to store data more compactly.

2.1 ROOT Format

ROOT is a scientific toolkit used for data processing, analysis, visualization and storage. The main framework is written in C++ and has interfaces for interactive usage, scripting, as well as compiled execution. It is the primary tool for high energy physics data analysis and is designed to handle the petabytes of data processed during collision experiments. ROOT provides object storage through a dictionary model with experiments exposing individual libraries to model and store components of their data.

2.1.1 Tree Model

The base unit of ROOT data storage is a `TFile`, a type of binary file that stores objects in a directory style. A `TFile` can store objects of any type, but the most common organizational structure stores a set of `TTree` objects. `TTree` objects are a collection of `TBranch` objects along with metadata. A `TBranch` is designed to store objects of a class, ranging from integers to floats to structured objects. Because ROOT is designed for fast I/O, `TBranch` objects are stored in a columnar format. That is, each `TBranch` has its entries stored consecutively on disk. For example, for an `TBranch` that stores 1000 objects of a `Event`(int, float, timestamp) class, the integer types will all be stored in a consecutive array on disk. This enables fast reading and writing of a single leaf of a `TBranch`. and better compression. To enable more efficient data processing, `TBranches` are subdivided into `TBaskets`, which allow compression and I/O operations to be done in smaller batches that work better with caches, and enable smaller chunks to be read from disk quickly.

ROOT has been designed to support native compression on disk. When data is added or modified, it is added to a buffer of memory space that is periodically written to disk. When the buffer fills to its capacity, objects are serialized using custom serializers and then compressed into a `TBasket`, as shown in Figure 2.1. Then, when data is read back into memory, decompression and deserialization recreates the original data. As ROOT has developed, more and more compression algorithms have been added to the library. As of version 6.21.00, ROOT supports Zlib, LZMA, LZ4, Zstd, as well as a backwards compatible Zip style algorithm. Each algorithm can be configured using a single parameter `level`, an integer between 1 and 9, which corresponds to the strength of compression, with a tradeoff of throughput and memory usage. The sizes of `TBasket` objects depends on user settings, system configurations, and objects sizes, and can be modified and adjusted for various computational goals. For instance, larger `TBasket` sizes enable better compression, but at the detriment of performance if the `TBasket` size is pulling more data into memory than is necessary for the analysis.

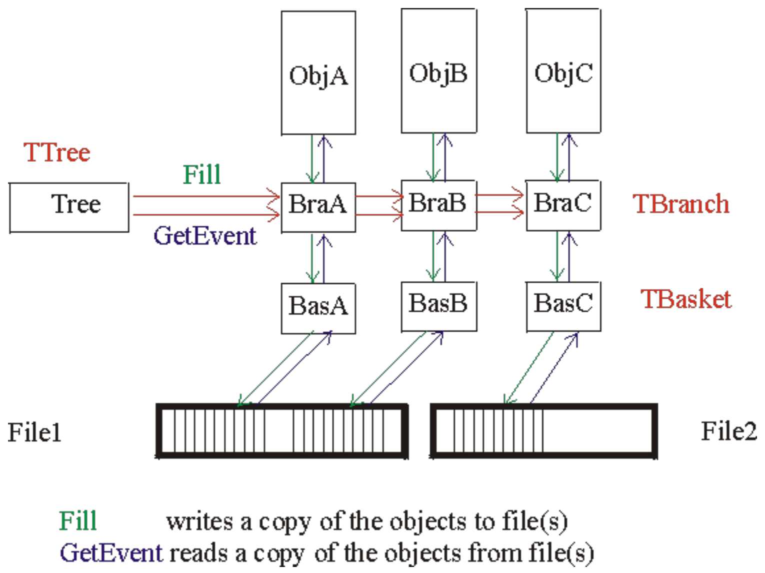

ObjA
ObjB
ObjC
TTree
Fill
Tree
BraA
BraB
BraC
TBranch
GetEvent
BasA
BasB
BasC
TBasket
File1
File2
Fill writes a copy of the objects to file(s)
GetEvent reads a copy of the objects from file(s)

Figure 2.1: TTree→TBranch→TBasket Structure

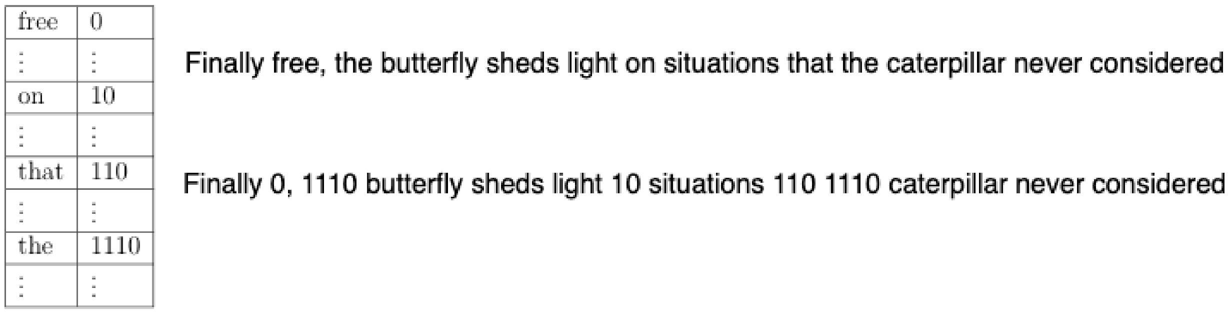
Source: `https://www-numi.fnal.gov/`

2.2 Data Compression

2.2.1 History

Data compression has been a central focus of media storage and data movement for the past 60 years. With the emergence of digital image and video storage, both lossless and lossy compression became essential to storing media data cost effectively. Entropy encoders, a type of compression scheme that operates on any data distribution, were one of the first compression techniques developed. A variety of compression techniques including arithmetic coding, range encoding and Huffman coding are forms of entropy encoding. These techniques can be divided into adaptive strategies, which change with respect to the data being compressed, and static strategies, called universal codes, which operate well on a fixed domain of data with a known probability distribution. These approaches work well on text, as a language can be efficiently encoded to shorter representations, as shown in Figure 2.2.

Two of the first modern entropy encoding compression algorithms are Shannon and Fano codes. Both techniques utilize a probabilistic model to convert source symbols into better sized prefix free symbols, reducing the size of highly probable sequences [51]. However, neither algorithm is guaranteed to produce an optimally short code, and therefore are seldom used in modern compression. Huffman trees, a successor to Shannon-Fano codes, are generated by a first pass through the data, creating a binary frequency tree, shown in Figure 2.3, which can then be used to rewrite data with optimally small new bit sequences [32]. This method differs from Fano's method by moving from the leaves to the root, rather than the root to the leaves [27]. This can be shown to be provably optimal if the data distribution across symbols is uniform and the probability of each symbol is a power of 2, but in typical data this is often untrue. Other methods of arithmetic coding are much more efficient on different sized symbols. These techniques use probabilistic distributions to limit the range of the output data, representing the data in an efficient base rather than a binary tree [57].



free	0
⋮	⋮
on	10
⋮	⋮
that	110
⋮	⋮
the	1110
⋮	⋮

Finally free, the butterfly sheds light on situations that the caterpillar never considered

Finally 0, 1110 butterfly sheds light 10 situations 110 1110 caterpillar never considered

Figure 2.2: Dictionary Coding for English Text

Later compression strategies use combinations of ideas from static and dynamic dictionary approaches paired with arithmetic coding techniques to rewrite data with prefix-free codes. In the 1970s, Ziv and Lempel developed a series of compression techniques which today still form the backbone of many compression algorithms. LZ77 uses a sliding window buffer as a dictionary to encode future data, allowing repetition to be stored compactly [60]. Each block is read individually and represented with a dictionary generated by the previous block, creating a simple model which works quickly. However, when data patterns repeat in longer intervals than the block size, this approach can perform poorly. LZ78 takes a different approach to dictionary compression, building up a running dictionary one character at a time. LZ78 enables longer string matching and larger dictionaries, increasing potential compression ratio [59]. However, LZ78 is much more resource intensive, and can unnecessarily store items in a dictionary long after they have appeared in the data.

The most popular successors to LZ77 and LZ78 include Lempel-Ziv-Storer-Szymanski (LZSS), Lempel–Ziv–Markov (LZMA), and Lempel–Ziv–Welch (LZW), which improve on the prior work and produce better compression algorithms for certain use cases. Another more recent approach, Context Tree Weighting (CTW), uses Bayesian modeling to predict future binary data using an online model. An average of models can then be computed, giving more weight to simpler predictors in order to arrive at a best model [56]. These techniques fundamentally rely on the same insight: previous data can be used to predict future data, and data can be rewritten to represent common sequences with shorter prefix free sequences.

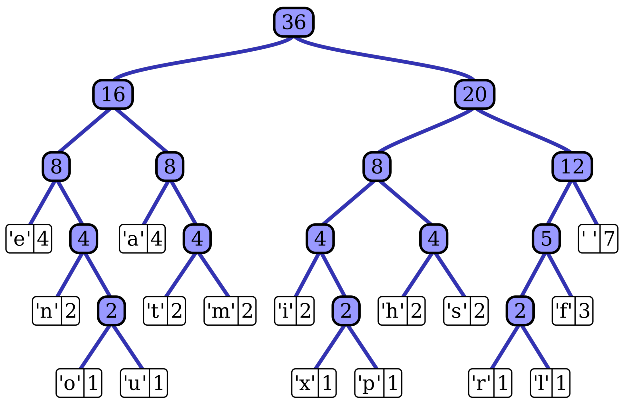


Figure 2.3: Example of Huffman Coding Tree

2.2.2 Current State of the Art

Modern compression strategies are often designed with a focus on a particular attribute: speed, compression ratio, ease of implementation, etc. We give a brief overview of 5 of the most popular general purpose compression algorithms: zlib, lz4, lzma, brotli, snappy, and zstd. Figure 2.4 demonstrates a variety of these configurations and their performance on a Mercurial dataset from the Firefox repository, consisting of a mix of code, text, and image data. Although this is not representative of a high energy physics dataset, it gives a rough idea of what performance looks like for these algorithms [52].

1. Zlib is an block based DEFLATE style algorithm, a variant of LZ77, that uses Huffman coding and was designed to work with PNG image data. This format is shared between zlib style headers and gzip style headers. Zlib is widely integrated into many operating systems as the system default compression. Zlib ranges in compression level from 0 to 9, where 0 is no compression and 9 has the highest compression ratio at the expense of speed [23].

2. LZ4 is also an LZ77 block based compression algorithm, but provides faster compression and decompression times than Zlib. LZ4 is used widely in compression oriented file systems and databases such as Hadoop, ZFS, and SquashFS where high throughput is crucial. LZ4 ranges in compression level from 0 to 12, where 0 is no compression and 12 increases compression ratio at the expense of speed [2].

3. LZMA also follows the LZ77 algorithm structure, but uses a significantly larger dictionary to take advantage of larger system memory in modern hardware. Additionally, LZMA implements larger contexts for its encoder model. LZMA is implemented in the 7zip compressed format and in the command line tool `xz`. LZMA ranges in compression level from 0 to 9, where 0 is no compression and 9 increases compression ratio on average at the expense of speed and memory cost [1].

4. Brotli is a streaming data compression algorithm which combines LZ77, Huffman coding, and context modeling. Brotli uses built in dictionaries which require offline storage to encode common strings, phrases, and byte sequences. Brotli limits window sizes to allow for device compatibility, and is widely supported by internet browsers and content distribution networks. Brotli ranges in compression level from 0 to 11, where 0 is no compression and 11 increases compression ratio on average at the expense of speed and memory cost [6].

5. Snappy is a block based compression algorithm designed for reasonable compression at very high speeds. Snappy runs significantly faster than zlib and is widely used in time critical database systems such as MongoDB and BigTable. Snappy is byte oriented and uses a fixed length dictionary, but does not use arithmetic coding. Snappy has only one level of compression but provides both block based and streaming protocols [29].

6. Zstandard, or zstd, is a block based compression algorithm that uses larger dictionaries

and minimum block size of 256 KB to make use of larger system memory. Zstd uses a LZ77 based dictionary along with Huffman coding and entropy coding through Finite State Entropy to achieve high compression ratio. Zstd is tuned by a compression level which can be adjusted between 0 and 22 for compression ratio and speed/memory usage tradeoffs. Zstd also introduced the capability to pretrain dictionaries, enabling better compression performance on small files [19].

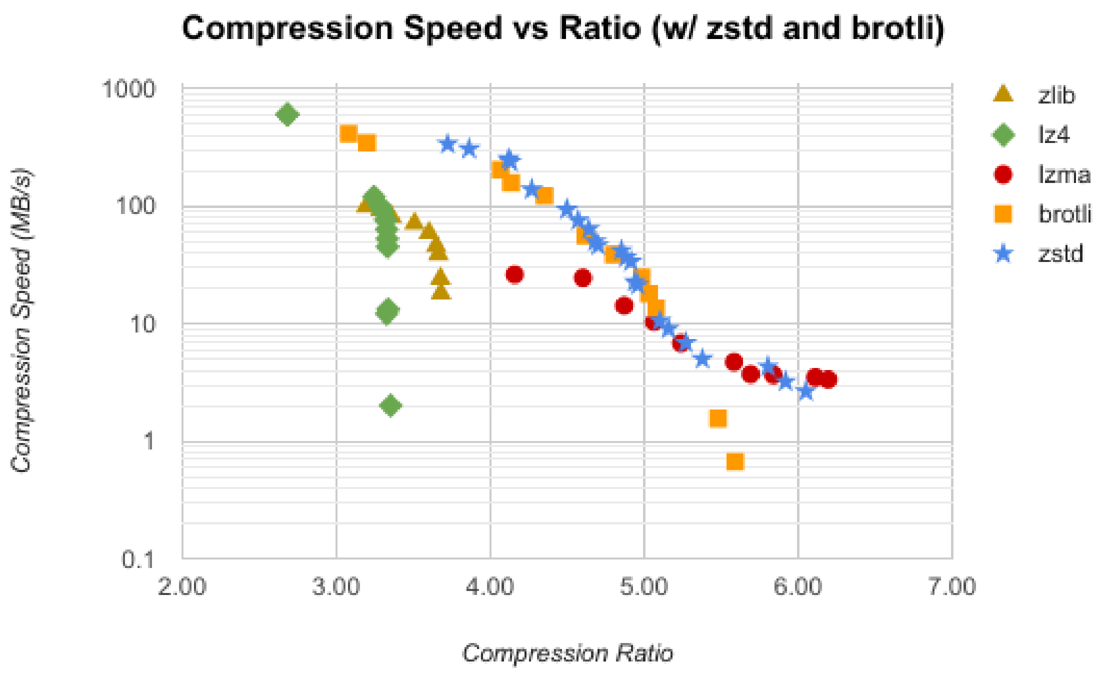


Figure 2.4: Comparison of Common Compression Algorithms

Image courtesy of [52]

2.2.3 Lossless vs. Lossy Compression

Although the default ROOT algorithms only support lossless compression options, one possible compression choice is to trade perfect reconstruction of data for an increase in compression performance. One strategy for implementing such lossy compression is to understand which portions of a dataset are not important and can be removed with minimal to no cost to the end results. For example, lossy floating point libraries often round values or truncate values to decrease the data size. This approach can be extremely effective in some scenarios, when future computations do not rely on the precision of the data. There are two main types of lossy coding: predictive coding and transform coding. Predictive coding uses a model that takes in previously decoded data to predict what the next byte or chunk of data will be, allowing only a correction to be stored. Transform coding projects the feature space of the underlying data to a lower dimensional space where data can then be quantized and be stored in reference to this new lower dimensional space.

Many formats for media storage utilize lossy compression to deliver incredibly small storage sizes at nearly indistinguishable appearance to a human eye. The JPEG format, a popular image format, rounds the precision of pixel data to effectively smooth the overall image. When the underlying data is understood well, these compression choices can be designed to fit the data by hand or through machine learning. However, when the data must be maintained exactly, lossy compression is not a feasible option as the reconstructed data will differ from the original data. Collision datasets require high data fidelity for proper analysis, and the cost to regenerate data is extremely high. Hence, in our analysis, we did not consider the use of a lossy compression algorithm as it would not be a practical solution for the vast majority of high energy physics data.

2.3 High Energy Physics Data Storage

2.3.1 Storage Requirements

The CERN compute network utilizes over 230,000 cores and more than 92,000 disks which have a total capacity of more than 280 petabytes of data. Additionally, about 32,000 tape cartridges are used to supplement storage capacity by an additional 400 petabytes. This

Medium	Price per GB	Lifetime (years)	Write Error Rate	Read Speed (MB/s)	Write Speed (MB/s)	Seek Time	Capacity per Unit
SSD	\$0.15	5	10^{-17}	2500	600	$50\mu s$	2TB
HDD	\$0.03	3	10^{-15}	200	200	10 ms	12TB
Tape	\$0.01	10+	10^{-19}	100	100	80s	15TB

Table 2.1: Comparison of Commercial Storage Media Products
[49] [12]

space is configured to store redundant copies of data, so the total data being stored is less than the total capacity 680 petabytes. The datacenters that host these storage racks and tapes are extremely costly to run, requiring electricity to power the devices when reading and writing, and energy to cool the buildings. More than 1.4 megawatts are used per year to cool the CERN datacenters. Each year, the amount of data generated and stored increases, so storage and energy costs are consequently increasing. In 2018, over 115 petabytes of data were added to permanent storage [16].

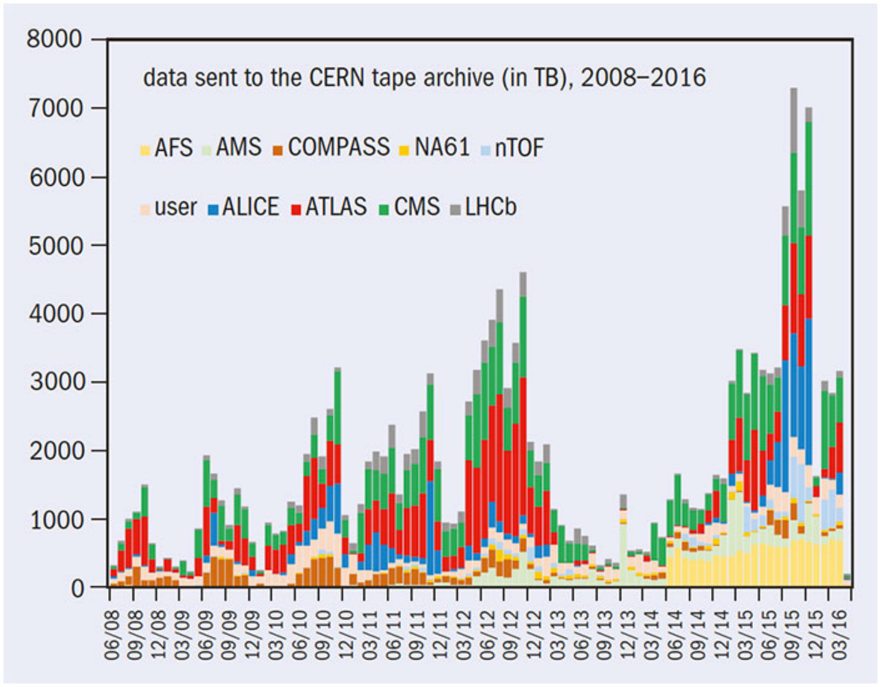


Figure 2.5: Increase in HEP Data Storage over Time

Source: `https://cerncourier.com/`

Both compute and storage scalability present challenges to the accelerating rate of data production for high energy physics research. As Figure 2.5 shows, data storage requirements have been steadily increasing over the past 10 years. Future projections by the HEP Software Foundation show data storage requirements surpassing current budget projections significantly in the coming years [7]. Figure 2.6 projects the storage requirements for the ATLAS experiment as it scales up for Runs 3 and 4 in the next 10 years. The model shows data storage requirements quickly outpacing budget constraints, necessitating both cheaper and more compressed storage.

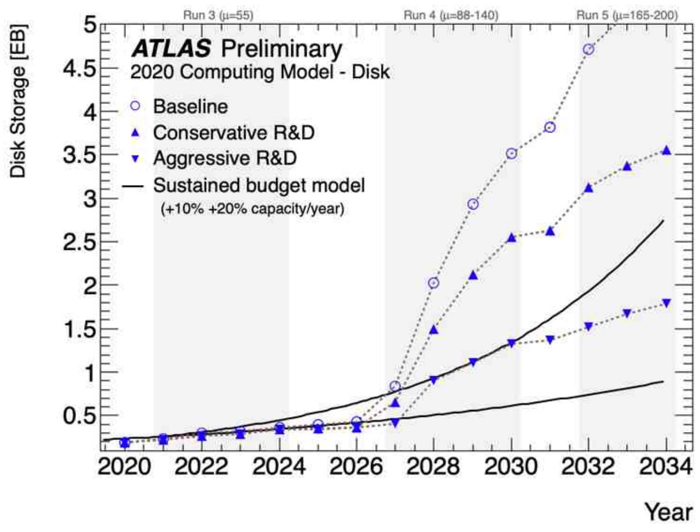


Figure 2.6: Projected HEP Data Storage Usage and Cost

Source: CERN Twiki - AtlasPublic Computing and Software

2.3.2 *Current Storage Techniques*

High energy physics data is almost exclusively stored in the ROOT format, composed of `TTrees` and `TBranches`. These files traditionally were stored in CERN Advanced Storage (CASTOR), but most data is being transitioned to EOS, where data is stored on hard disks. This storage is primarily used for remote analysis and fast processing, and has very low latency compared to CASTOR, which uses a combination of hard disks and tape [24].

Users connect to the servers using the `xrootd` protocol, a distributed infrastructure which provides resiliency across geographically distributed datacenters. `xrootd` enables users to access files with high reliability without knowing exactly where an individual file is located. This protocol is mainly utilized for read only jobs, and provides very fast and simple file access.

RECO and AOD ROOT files are typically stored in a compressed format, usually with the algorithm Zlib [50]. This is a basket level compression, and data is decompressed as needed when reading or running analysis. According to CERN, the LHC Tier 0 computing grid runs around 1 million jobs per day, transmitting data at speeds up to 10 gigabytes per second. Because of the massive scale that these storage clusters operate at, even at peak transmission rate, only a small fraction of the total CERN data storage is being accessed in any given day. We see that a one size fits all approach for data compression and storage is not optimal for providing the best performance in a variety of circumstances.

CHAPTER 3

EXPERIMENTS

In this chapter, we discuss the types of data found within CMS and ATLAS ROOT files, and provide an overview of techniques used to compress data. We also describe the experimental setup and characterize the datasets used for evaluation.

3.1 Dataset Selection

For our analysis, we use a combination of experimental and Monte Carlo simulated data from the ATLAS and CMS experiments. This data is representative of the data stored in the CERN grid, so we can use it to validate our compression approaches and project performance across the entire CERN storage network. These datasets have experiment specific objects and types which makes software versioning tricky as the latest software releases for the experiments do not always support the newest versions of ROOT.

3.1.1 CMS Higgs Boson Datasets

We use a selection of data from CMS entitled “Higgs-to-four-lepton analysis example using 2011-2012 data” to provide a representative sample of AOD files[34]. CMS is one of the largest experiments run at the LHC in the past 10 years, and many petabytes of data from it are stored in the CERN grid. This dataset represents the largest and best public example of the data stored for the CMS experiment. The released dataset contains a reproducible example of the Higgs-to-four-lepton decay channel that contributed to the Higgs boson experimental discovery in 2012. It represents a combination of Monte Carlo simulations and legacy data that was used in practice for many CMS publications. The dataset and accompanying paper have been cited more than 10,000 times in the past 8 years. Because of CERN data release regulations, this data only comprises approximately 50% of the total data used in the analysis but still provides statistically significant data to reconstitute the result and is the most complete and representative CMS dataset available for public use.

The CMS dataset contains 21 different subdatasets that that range in size from 100 gigabytes to 11 terabytes, and are divided into between 15 and 10000 files. These files are in the ROOT format and contain validated runs. Each ROOT file contains a selection of `TTree` objects, named Events, LuminosityBlocks, MetaData, ParameterSets, Parentage, Runs. The vast majority of data is stored in the Events Tree, which contains 98% of the total data size. These Event Trees contain between four and five thousand branches and subbranches of various types and sizes.

Filename	Trees	Total Branches	Total Size (MB)
00A0AA1B	7	4798	61
02E4E236	7	4798	194
0AEADC9B	7	5060	3770
0C7108A5	7	4798	2400
0E74BA28	7	4767	2696
16EF3E2F	7	5060	3842
1C129990	7	4767	4004
265CDD9C	7	4792	2928
28FE3EB7	7	4798	2738
2CD32440	7	5060	3817
2E598EF0	7	5060	3709
3042A96D	7	4792	4062
322EB608	7	4798	2236
3CFA06BB	7	4767	3802
3E24CF15	7	4798	3642
48445042	7	5091	2151
4A314A97	7	4767	2520
4C849F63	7	4798	2554
4E351466	7	4798	2819
62A338BF	7	4792	4068
740100D4	7	5060	3717
746ECCD4	7	4798	3281
801EAD1E	7	5060	3894
B0167B1E	7	5091	2124
B627133A	7	4792	3920
B8BF192D	7	4798	68
C8BDE2B2	7	4767	3716
CACA7DE4	7	4798	91
F852E7F0	7	4798	89
FC9DC0F7	7	4798	69

Table 3.1: CMS Test Files

It was computationally infeasible for us to evaluate the compression techniques on every file in the 70 TB dataset. However, because each file contains statistically similar data, and is stored in the same fashion, sampling a set of data randomly gives representative data for the entire collection. We randomly sampled 30 files comprising 78 GB from the entire collection of 48,000 files (70 TB). These file range in size from 61 megabytes to 4 gigabytes.

We show a summary of the CMS files used for our analysis in Table 3.1.

3.1.2 ATLAS Experimental Datasets

We also use a sample of ATLAS data used in experimental workloads to evaluate our compression approaches. The dataset consists of 3 different parts. First, there is a set of DAOD (Derived AOD) of real experimental data generated at CERN. Second, there is a set of Monte Carlo simulated DAOD data. These two collections are representative of the approximately 80PB of DAOD data stored by the ATLAS experiment in CERN datacenters. Finally, there is a set of plain ROOT ntuple files that represent typical end user's analysis data. Taken together, these files are representative of the data present in the ATLAS data storage system and can provide a good idea of what our compression approaches on the entire ATLAS storage system would provide.

The ROOT files range in size from 250 MB to 16 GB, and store between 15 and 100 `TTree` objects, along with other metadata and histograms. The trees have up to 1000 branches each. This 85 GB selection is a small representative sample of ATLAS AOD files and provides a good idea of how compression techniques would preform on larger sets of ATLAS data. We show a summary of the ATLAS files used for our analysis in Table 3.2. We obtain the samples from the University of Chicago Enrico Fermi Institute's Tier 2 grid.

Filename	Trees	Total Branches	Total Size (MB)	Filename	Trees	Total Branches	Total Size (MB)
3.05630052.01	10	1198	1787	7.16395000.11	10	1110	1738
3.05630052.01	10	1198	1787	7.16395000.12	10	1110	1695
3.05630052.01	10	1198	1787	7.16395000.13	10	1110	1692
3.05630052.01	10	1198	1787	7.16395000.14	10	1110	1754
3.05630052.01	10	1198	1787	7.16395000.15	10	1110	1707
3.05630052.01	10	1198	1787	7.16395000.16	10	1110	1691
3.05630052.01	10	1198	1787	7.16395000.17	10	1110	1750
3.05630052.02	10	1198	1613	7.16395000.18	10	1110	1778
3.05630052.02	10	1198	1613	7.16395000.19	10	1110	1635
3.05630052.02	10	1198	1613	7.16395000.20	10	1110	1676
3.05630052.03	10	1198	5375	7.16395000.21	10	1110	1661
3.05630052.03	10	1198	5375	7.16395000.22	10	1110	1712
3.05630052.03	10	1198	5375	7.16395000.23	10	1110	1697
3.05630052.04	10	1198	5377	7.16395000.24	10	1110	1746
3.05630052.05	10	1198	5369	7.16395000.25	10	1110	1691
3.05630052.06	10	1198	5368	7.16395000.26	10	1110	1673
3.05630052.07	10	1198	5365	7.16395000.27	10	1110	1700
3.05630052.08	10	1198	5373	7.16395000.28	10	1110	1666
7.16395000.01	10	1110	348	7.16395000.29	10	1110	1737
7.16395000.02	10	1110	343	7.16395000.30	10	1110	1738
7.16395000.03	10	1110	307	7.16395000.31	10	1110	1629
7.16395000.04	10	1110	287	7.16395000.32	10	1110	1713
7.16395000.05	10	1110	323	7.16395000.33	10	1110	995
7.16395000.06	10	1110	307	u.f.18935005.01	101	44157	4072
7.16395000.07	10	1110	301	u.f.18935005.02	101	44157	390
7.16395000.08	10	1110	266	u.f.18935006.01	101	44157	3561
7.16395000.09	10	1110	285	u.f.18935006.02	101	44157	376
7.16395000.10	10	1110	279	u.f.18935006.04	101	44157	895

Table 3.2: ATLAS Test Files

3.2 Primitive Types

Primitive types are the basic underlying data types provided by C++. ROOT and the libraries that run on top of ROOT for analysis support these types, and build up more complex objects and structures by composing them in various ways. C++ supports `int`, `char`, `float`, and `double` types. We also consider unsigned versions of `int` datatypes. Other primitive types are derived from these 4, as `boolean` types are simply 1 byte integers restricted to 0 and 1, and `int` and `uint` types can be represented with 16, 32, or 64 bits. Every type can be extended into an array of that value, where the data of the same is simply stored consecutively for a given length in memory. We focus on the `int`, `float`, `double`, and `boolean` types as they comprise the majority of high energy physics data, and therefore present the greatest opportunities to increase overall compression. ROOT maintains a set of machine independent types, so these types will always be the expected size. The CMS and ATLAS libraries heavily utilize large structured objects to store data, but many of these objects are simply a collection of primitive datatypes and can be stored as such. Our approach to data compression for these types can be extended to any object, but we focus on primitive types in this study. Even though each of these types appears as a un-typed string of bits to a compression algorithm, with the knowledge of the underlying data type, we can use information about data distribution to better represent the data for compression.

We view all structured object data as a collection of individual primitive types. If a structure is of the form `Event`(int, float, boolean), we compress the int, float and boolean pieces separately, even if ROOT compresses the `Event` object directly. This is an application of columnar storage, as shown in Figure 3.1. This approach collocates similar data, enabling compression algorithms to better recognize data patterns with a smaller window. Furthermore, we flatten all arrays and higher dimensional structures. For n-dimensional arrays, only the size of the dimensions are necessary to reconstruct the full structured data, so we can just store a small number of additional integer values. However, for arrays with irregular

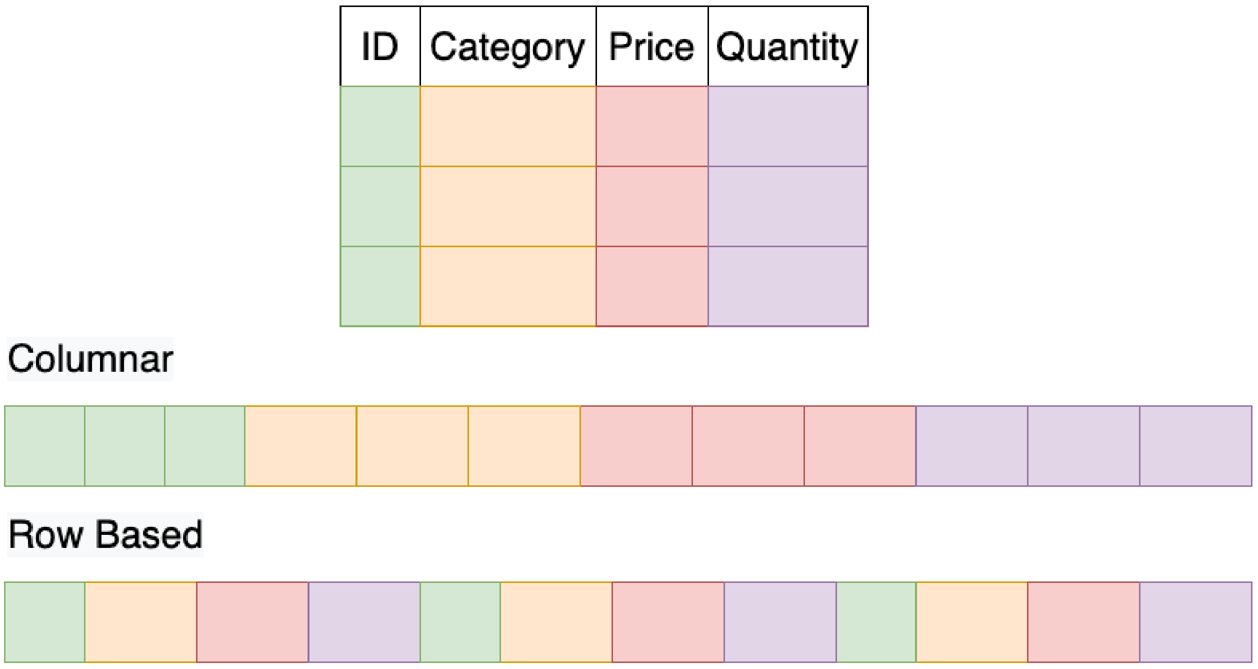


Figure 3.1: Columnar vs Row Storage

structure, we need to store more information to reconstruct the “shape” of the data.

High energy physics uses jagged arrays (arrays whose elements are differently sized subarrays) to store variable length data [42]. These jagged arrays allow variable length data to be stored in a compact fashion as each subarray only has to be as large as the data it needs to fit inside of it. When written to disk, this data must be stored with some metadata to determine where each subarray begins and ends. One method for storing this data is through an offset column, which is a secondary integer column that stores the length of each subarray. This creates storage overhead for one column, as an integer offset column with length equal to the number of subarrays must be stored. However, in many cases, multiple jagged arrays are indexed by the same offset column, so only one copy needs to be stored. ROOT files store the offset column in its own branch, and it is the only data necessary to reconstruct the jagged array. Figure 3.2 demonstrates this layout, as the branch on disk is stored as 10 consecutive integers, but when analyzed can be viewed as having 4 elements, each of which is an integer array of length between 1 and 4.

We consider approaches that utilize the inherent structure of the jagged array to rearrange data for compression. For example, in Figure 3.2, we attempt to compress the data as it traditionally resides on disk and in a column based orientation. In the traditional storage of jagged arrays, data is not aligned by column, but by row. If the element in the first position of each array stores a height, and the second position stores a speed value, the elements in the first index position are much more likely to have bit level similarity, and so would be compressed better by an approach that arranges the data by index.

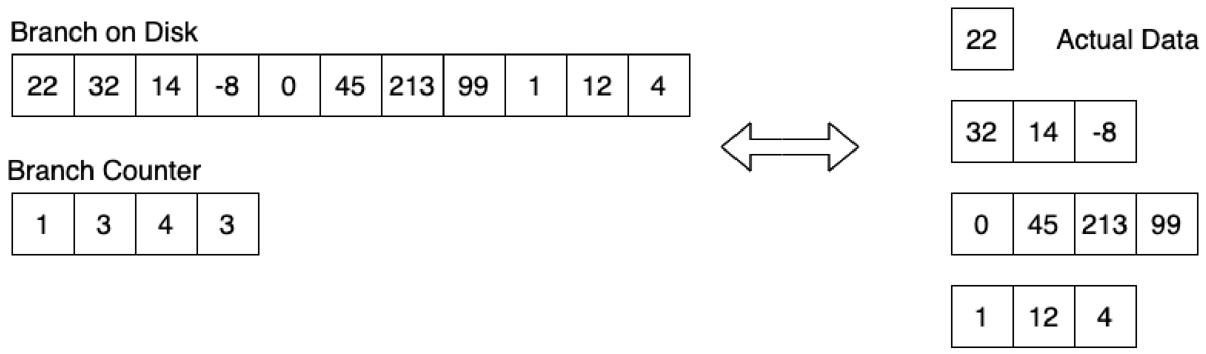
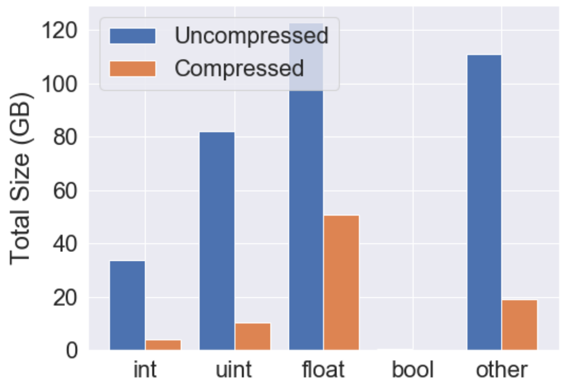
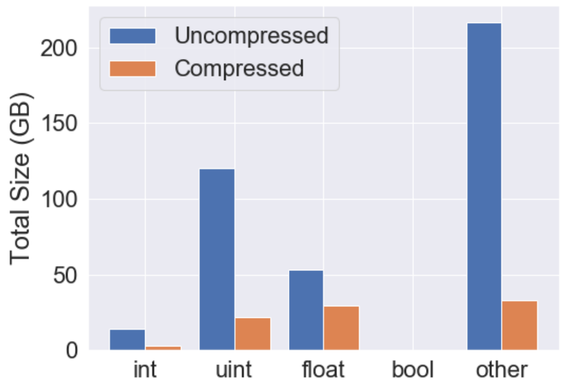


Figure 3.2: Jagged Array



(a) CMS Data



(b) ATLAS Data

Figure 3.3: Size of Uncompressed and Compressed Data by Datatype

3.3 Metadata and Structured Objects

To enable fast summary statistics and quick analysis, ROOT stores metadata per `TFile`, `TTree`, and `TBranch` as well as with many smaller objects. This metadata consists of the type of the object, the size and shape of the object, compression algorithm, compression configuration, basket sizes, as well as some summary statistics for the branch. Across the CMS and ATLAS files, this metadata adds an additional 1-5% of storage space to a `TBranch`, but the size of the metadata can vary significantly based on the object type and settings.

ROOT also enables users to store additional metadata as `TObjects` attached to any branch or tree, allowing a user to cache important data needed for analysis, or structure needed to plot objects. The amount of metadata stored with an object ranges from file to file, and from branch to branch. ROOT metadata can be compressed with the branches and trees, and is decompressed and deserialized when a file is read using ROOT. ROOT metadata is structured, but its variability makes it difficult to create specific compression approaches beyond applying a general purpose compression algorithm. As shown in Figure 3.3, for CMS and ATLAS ROOT files, metadata and other structured objects make up more than 30% of the uncompressed data. However, this data has irregular structure and therefore provides few opportunities to use underlying data structure to improve compression. Hence, we focus mainly on the integer and floating point datatypes, which make up more than 50% of the compressed representation.

3.4 Data Aggregation

From our analysis of compression algorithms, we see that algorithms use similarity within a block, or across two consecutive blocks, to rewrite data in a more compact representation. The more data we provide in a block, therefore, the better the algorithm can extract similar data patterns. Hence, we expect that larger branches, which can fill up a block-based compression algorithm's window, and provide good history for pattern matching, would lead to better compression performance. As Figure 3.4 demonstrates, we see that larger branches have higher compression ratios. Data aggregation is a potential way of increasing the compressibility of smaller data columns. Our approach takes this insight into account in two ways.

First, we compress branches, not baskets, allowing the compression algorithm to view the entire branch at once. Higher levels of `zstd` and `zlib` compression can have window sizes of more than 256 MB, whereas the basket sizes used in the CMS and ATLAS files range from 16 KB to 1 MB. By aggregating the data prior to compression, we give the compression algorithm the best view of the data.

Second, we also evaluate aggregating all branches of a given name across the ROOT files of an experiment. We look through all 30 of the CMS files and combine identically named branches to increase the base size of each branch file by up to 30x. Although the data stored in each file varies slightly, the vast majority of branches are found in each file. We find 5,790 individual branch names across the 30 files, and see that each individual file stores between 82 and 90% of these branches.

Because many of the branches are smaller than 256KB individually, if we can aggregate 30 branches that store the same data, we can fill the algorithm's compression window and get the full benefit of the compression window size. This approach will provide the most benefit when the aggregated files have a significant degree of similarity between them. In our experiments, we sample a set of CMS files, so the event and time based similarity may be lower than if the files were consecutively recorded. In a large scale implementation, data would be aggregated with files that were recorded before and after it, and could have a higher degree of similarity across branches of the same name.

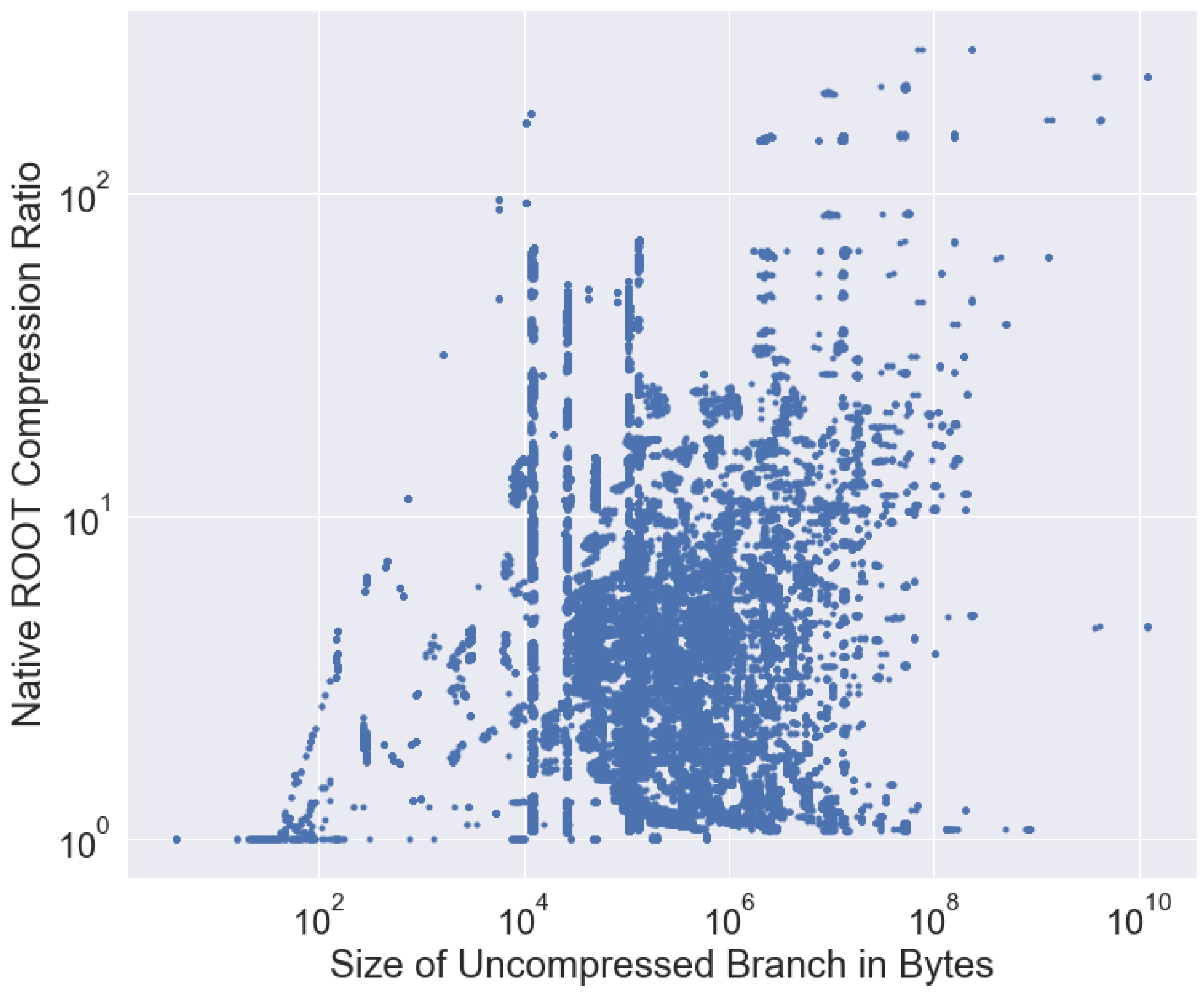
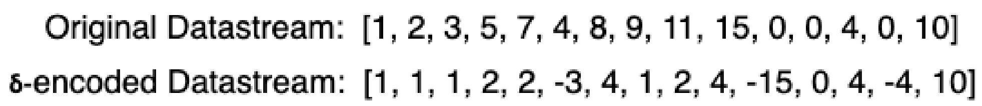


Figure 3.4: Ratio of Compression Ratio to Branch Size (ATLAS+CMS)

3.5 Compression Techniques for Known Datatypes

3.5.1 Delta Encoding

Delta encoding, also known as data differencing, is a technique used to store the delta, or change, between sequential objects. In the context of compression, delta encoding can simplify the bit representation of a series of objects, leading to better compression. For example, on an integer sequence, we can delta encode and take the difference between sequential values, as shown in Figure 3.5. While this does not directly reduce the size of the data, it



Original Datastream: [1, 2, 3, 5, 7, 4, 8, 9, 11, 15, 0, 0, 4, 0, 10]

δ-encoded Datastream: [1, 1, 1, 2, 2, -3, 4, 1, 2, 4, -15, 0, 4, -4, 10]

Figure 3.5: Delta Encoding

Type	Original	1st Delta	2nd Delta	3rd Delta
Int	6.31	8.42	7.47	7.07
UInt	11.72	10.19	9.73	9.46
Float	2.14	1.68	1.69	1.66

Table 3.3: Compression Ratio of Delta Differencing Approaches

can increase the repetition within the data, which compression algorithms can more readily compress. This process can be done in one pass, and is very efficient. For example, a linear sequence of integers, which has very little bit level similarity, can be delta encoded to a start value, and then a series of steps, all of which have the same bit level pattern. To reconstruct the original series, we just need a second linear pass to sum the delta encoded stream up to each index. We define a second level delta passes as a delta pass on the output of the first delta pass. Higher levels are defined similarly.

We evaluated this technique on all primitive datatypes, but found that it provides minimal to no benefit to floating point and unsigned integer values. Floating point values are encoded differently than integers, and we find that subtractive differencing is not effective and can cause data loss due to floating point imprecision. Boolean values will simply be flipped by a delta pass, so this approach provides no benefit. Unsigned values provided an interesting challenge, as they cannot be directly delta encoded due to sign issues, as a negative difference cannot be stored in an unsigned integer. However, we can either interpret the underlying bits as a signed integer, or set an implied highest order bit for subtraction to allow the data to be stored correctly. For completeness, we considered other bitwise operations such as exclusive or, and multiple levels of difference in a preliminary study on a sample of ATLAS data. We show the results in Table 3.3, and see that the first difference for integer data is the only type of delta pass that shows improvement.

3.5.2 Float Splitting

The IEEE floating point specification, as presented in Figure 3.6, uses a sign, exponent, and fraction to represent floating point values. Unlike with integers, floating point values that are close together by the standard distance metric can have a large Hamming distance when viewed as bitstrings. This presents a challenge for data compression, as regular bit level patterns are what enables good compression. Signle precision floating point values have 1 sign bit, 8 exponent bits, and 23 fraction bits. For floats that are close together in value, the exponent and sign will often be similar, but the fraction can be very different. We reorganize the data to align the sign with the fraction and store them together in a array of 24 bit values, and store the exponent in an 8 bit array of values. Then these arrays can be compressed separately, potentially enabling better compression.

For 64-bit double values, this reorganization is not as simple, as the data does not seamlessly pack into byte-sized values. We evaluate adding an additional 4 bits of padding to the combined 52 bits of fraction and sign to make it 56 bits = 7 bytes, and adding 4 bits to the exponent to make it 16 bits = 2 bytes. This padding allows CPUs to operate more efficiently as bit level operations cannot be done on standard architectures. Hence, working with non-byte aligned values would require multiple read/write cycles per value. We also evaluate the splitting pattern without the padding for completeness. This technique, like delta encoding, provides no direct benefit to storage size on its own, but paired with a compression algorithm, can improve compression ratio. We also consider delta encoding style techniques for the split floating point values, using exclusive or, first difference, and other bitwise operations.

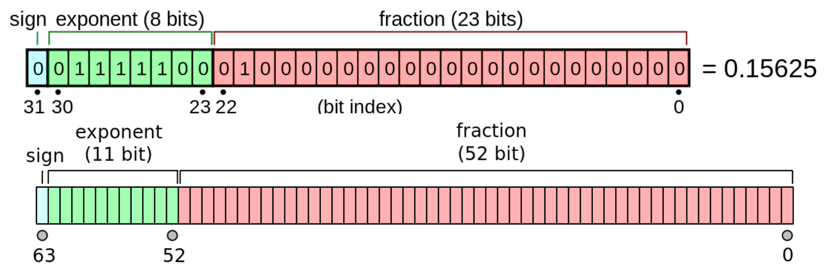


Figure 3.6: IEEE Single and Double Precision Floating Point Specifications

3.6 Methodology

3.6.1 Extracting Data

ROOT is designed for enabling complex data analysis and plotting, but is not optimized for data export. Newer versions of ROOT support DataFrames and more data compatibility with other formats such as Apache Parquet, but are not fully compatible with the CMS and ATLAS software. Additionally, because most ROOT workloads involve using a specific file with a known structure, automating extraction or analysis with ROOT is quite complicated. Objects can be stored with unknown padding, alignment, and serialization, making extraction a challenge.

We use the Python package `uproot` to extract most of the primitive datatypes. This package allows users to access ROOT files both in an interactive and scripted fashion. However, `uproot` does not support most of the non-standard objects used by CMS and ATLAS, so only primitive datatypes can be extracted using this method. We iterate through the trees, and types of branches, and flatten arrays to export all primitive datatypes into a `numpy` format, which can then be written to a file as the uncompressed branch. We also track the metadata, and the size of the metadata associated with each branch to ensure we can reconstruct the file in full. Even though it cannot handle all objects, `uproot` is significantly faster than using ROOT macros to extract data, so we use the two approaches in tandem.

We then use ROOT to extract all branches with structured format not readable by `uproot`. This only represents a portion of the remaining branches, as we do not extract many of the table, vector, and variable length types. We attempted to extract these types, but found that the individual storage structure of the objects in each file were nonstandard, and made it difficult to ensure that all data was being correctly extracted. We loop through the branch, copying the disk representation of each uncompressed object to a buffer, and then writing the buffer to disk at the end of the branch. We store the data from each branch in little-endian format in an individual file, and track the native ROOT compressed size, uncompressed size, metadata size, and other compression related metadata.

3.6.2 Experimental Setup

We evaluate the compression performance on a server running Scientific Linux 7.8. We evaluate with a 2.8 GHz 8-core Intel Xeon E5440 processor. The server has 16 GB of RAM, and 32 KB of L1 cache. The versions of all software packages used are shown in Table 3.4. All of these versions are configurations from an up to date ATLAS Tier2 server at the University of Chicago. For the experiments contained in Chapter 5, we evaluate using a 2-core Intel i5 processor to allow us to test using ROOT 6.21.00. We do not compare across the two platforms.

We use C++ to run compression tests on the extracted files, using the C++ APIs provided by `zlib`, `snappy`, and `zstd` to compress each file. We use single threaded versions of each compression algorithm, and compress and decompress with default settings unless otherwise indicated.

Experiment	Package	Version
CMS	CMSSW	6.2
	ROOT	5.34.97
ATLAS	ROOT	6.04.16
	AnalysisBase	2.5.1
Shared	xrootd	4.11.1
	Python	3.6.8
	numpy	1.18.1
	uproot	3.11.3
	snappy	1.1.18
	zlib	1.12.11
	zstd	1.4.5

Table 3.4: Software Versions Used

3.7 Data Compression Strategy Implementation

3.7.1 Compression Algorithms

We select the `zstd`, `zlib`, and `snappy` algorithms for evaluation. `zstd` is a modern, highly configurable compression algorithm designed to make use of larger system memories and faster clock rates. `zlib` is the standard used in most high energy physics ROOT files, and is a general purpose algorithm designed for performance in a variety of situations. We also evaluate `snappy`, a throughput focused algorithm which does not use arithmetic coding to improve speed.

3.7.2 Configurable Parameters

We evaluate the compression algorithms settings, considering both the compression level and window size for `zlib`. We run `snappy` in default block mode, and `zlib` at levels 1, 5, and 9. Higher levels for `zlib` and `zstd` use more memory and computation to store larger dictionaries, often leading to better compression ratios. We vary the block size parameter for `zlib` between 8, 12, and 15, which sets the block size to 256, 4096, and 32768 bytes. In preliminary evaluations, we experimented with the many configurations of `zstd`, but found that the best way to achieve good performance was through the auto-configured levels, which has preset configurations for parameters like window size, memory usage, and dictionary styles.

3.7.3 Compression Performance Evaluation

Metrics for Compression

- Compression Ratio: $\frac{\text{Uncompressed Size}}{\text{Compressed Size}}$, reflects how compressed data is from the original.
- Compression Throughput: $\frac{\text{Uncompressed Size}}{\text{Time to Compress}}$, shows single core throughput.
- Decompression Throughput: $\frac{\text{Uncompressed Size}}{\text{Time to Decompress}}$, shows single core throughput.

We clarify that when we report a compression ratio for multiple files, we compute the total compression ratio by taking the total size of uncompressed data and dividing it by the total size of all data after compression. We choose this metric as our data samples are representative, so their relative sizes are important. Therefore, a geometric mean or arithmetic mean would represent a skewed metric.

We find that conveying improvements in compression ratio can be abstruse, so we clarify our meaning here and use only this meaning in the rest of the thesis.

- If we start with a 30 GB file and compression Approach A has a compression ratio of 2, and compression Approach B has a compression ratio of 3, we say "compression Approach B produces a 50% improvement in compression ratio". We also could discuss the reduction in data size, and would say "Approach B leads to a 33% reduction in resulting data size as compared to compression Approach A".

One challenge presented by tracking resource utilization for compression algorithms is resource management by the operating system. Setting an algorithm to run at high compression levels can enable the use of dictionary sizes beyond the available RAM, in which case the dictionary size would be artificially restricted by the system resource constraints. The server used for evaluation only has 16 GB of RAM, so the results presented in the next section are a conservative estimate of compression results and represent a lower bound for scenarios where more than 16 GB of RAM are available during compression.

We do not measure the RAM usage of the compression platforms, but use the built in memory estimators to compute the total memory for levels 9 and below and find that memory consumption does not exceed 0.75GB. This clearly fits within the memory baseline of 2GB per core for CERN clients. However, `zstd` has a flexible memory policy on levels 19 and above, and so we cannot provide exact memory numbers for these levels. We note that compression and decompression with levels greater than 19 would only be done on data center servers in a large scale implementation. To mitigate the interference of multiple processes fighting for RAM and causing performance degradation, we compress using only one process at a time, but on a server with greater resources, each core could process in parallel.

3.7.4 Implementation of Delta and Float Splitting

We use a single pass over the data to implement the delta encoding pass in place. Because splitting floating point values requires the creation of two different sized arrays, it cannot be done in place, but is still done in one pass. We allocate two arrays: one for the sign and mantissa components, and one for the exponent components. Then, we use bit manipulation to extract each portion of the floating point value and aggregate them into each array. These techniques only require simple arithmetic and bit operations, can be implemented in fewer than 10 lines of C code, and can be done in a single streaming pass. We can also compose multiple techniques within the same linear pass. This passes could be integrated into ROOT or other software packages with low engineering and performance costs.

3.7.5 Pretrained Dictionary Compression

Most modern compression algorithms utilize a combination of dynamic and adaptive dictionaries to achieve good compression performance. However, this performance degrades when proper context cannot be achieved due to small files. Although aggregating baskets to compress an entire branch at once mitigates this challenge to some extent, many complete branches are still much smaller than `zstd`'s buffer of 256 KB. `zstd` includes an option to pretrain a dictionary for improved compression on small files. Because the branches often have similar distribution across the different files in a dataset, we hypothesize that pretraining the algorithm on the branches in one file will aid compression performance when used on other files.

It would be cost prohibitive to create a trained dictionary per branch, so we aggregate all files of a given type and create a single, shared dictionary. Hence, we have a pretrained dictionary for each size of every primitive type, and then dictionaries for other commonly used types. This adds the cost of generating and storing the dictionary, but we keep the dictionary size capped at the default 110KB. Dictionary size represents a negligible percentage of the total stored data, so we can still increase the compression ratio. We evaluate this trained dictionary on the files it was trained on, as well as other files from the same experiment not used for training to get cross validation of our technique.

3.7.6 File Aggregation

To implement aggregation across files, we simply combine branches of the same name into an aggregate branch which stores the data from all files. Then we run the exact same compression workloads, compressing as if the data was from one large file. We find that some of the larger branches (>100 MB) present a challenge to compress when aggregated across 30 or more files, so we set a threshold to cap the size of each aggregate file at 500 MB. If the aggregate data would be larger than 500 MB, we create multiple aggregate files.

Files larger than 500MB require more than the available RAM to compress at high levels of `zstd`. This group of files is quite small (fewer than 0.1%), and already would fill windows for compression, so we find that aggregating them would provide minimal benefit to compression ratio.

CHAPTER 4

EVALUATION

We evaluate the compression approaches discussed in Chapter 3 to understand how they perform on high energy physics data, both in terms of compression ratio and compression throughput. We also characterize the decompression performance for these algorithms and techniques. Most analysis workloads are read heavy, so the decompression throughput affects the total run time of an analysis workload. Compression can be done when resources are available, as it is not required to access the data, only to reduce the data size.

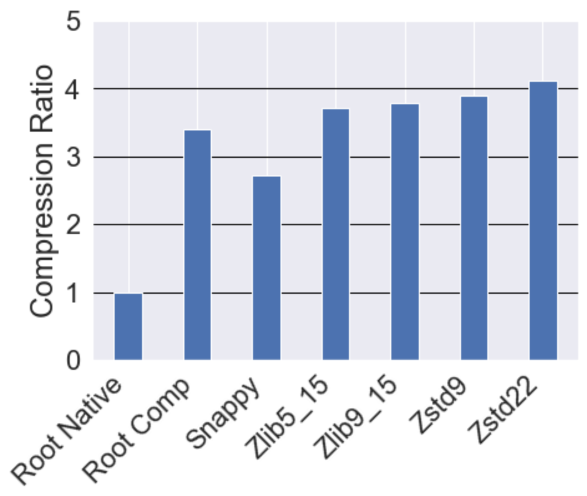
4.1 Compression Ratio

We begin our analysis by looking at how the various algorithms, configurations, and additional passes affect the compression ratio of the data, which we defined in Section 3.7.3.

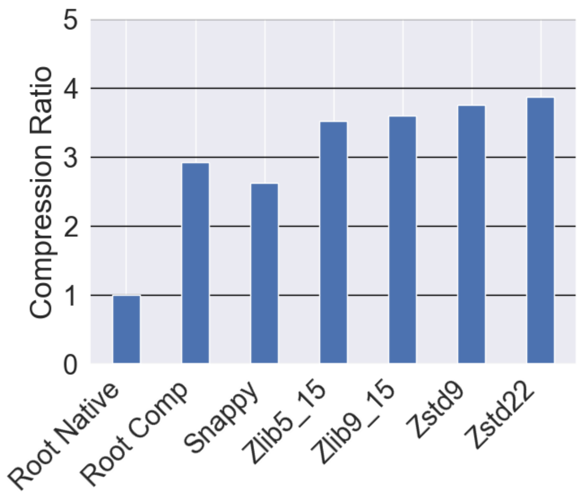
4.1.1 Algorithm Comparison

We evaluate the algorithms `zstd`, `zlib`, and `snappy` to compare their compression performance to the `zlib` compression used currently by the CMS and ATLAS datasets. Both experiments have their default compression set to `zlib` with level set to either 1, 5 or 7. In Figure 4.1, we present the compression ratio for each algorithm on extracted branch level data. We display all compression ratios relative to the total size of all branches, along with their branch level metadata as "Root Native". "Root Comp" (Root default compression) represents the ratio of all branches and their branch level metadata as the branches are currently compressed in a ROOT file.

In Figure 4.1, and in all other diagrams, the first number after an algorithm represents the level at which the algorithm was run, and a second number, separated by an underscore, represents the compression window setting. If no second number is denoted, the default



(a) CMS Data (Table 3.1)



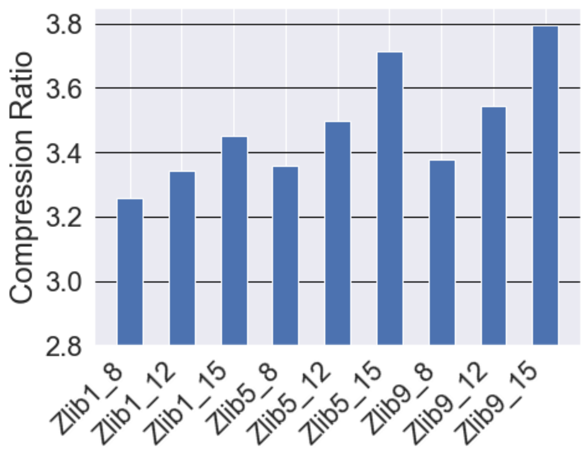
(b) ATLAS Data (Table 3.2)

Figure 4.1: Compression Ratio Across Algorithms

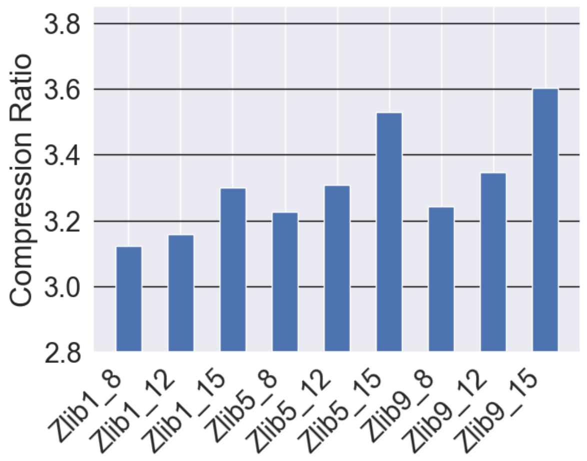
window setting is used. The compression ratio is computed by taking the sum of the total uncompressed size of all data extracted from the ROOT files, and then dividing by the total size of the compressed data as described in the metrics section of Chapter 3. As described in Chapter 3, this figure does not include the file level metadata, or other types we did not evaluate. A full evaluation of the overall effect on the resulting file size is presented later in the chapter. We obeserve that `zstd` with level 9 and 22, and `zlib` with level 5 and 9, can outperform the compression done within a ROOT file.

We find that the performance of the algorithms is relatively constant across the ATLAS and CMS datasets, but the CMS data appears to be better compressed as the compression ratios are 10-15% higher for every evaluated algorithmic configuration. Additionally, `zstd` with level=22 provides the best compression ratio across the board. We elide `snappy` from most other results after this point as it fails to be competitive with the currently used compression, and therefore represents a poor candidate for improving data storage.

In Figure 4.2 we present a more thorough look at how window size affects compression ratio performance. We consider `zlib` with 3 different settings for level and 3 settings for window size, ranging from the smallest to largest window sizes. With a window size set to 8,



(a) CMS Data (Table 3.1)



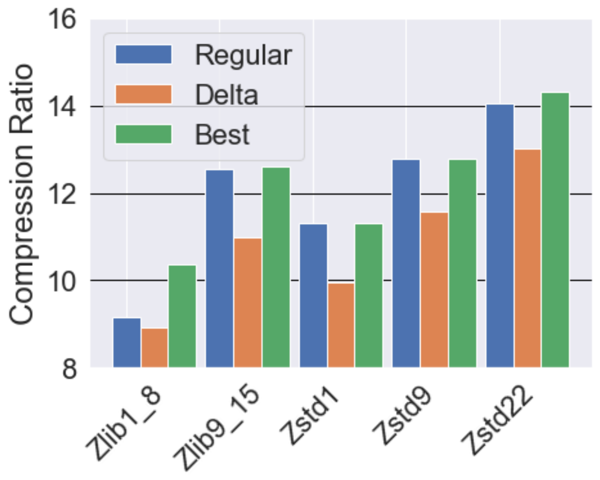
(b) ATLAS Data (Table 3.2)

Figure 4.2: Compression Ratio Across Zlib Configurations

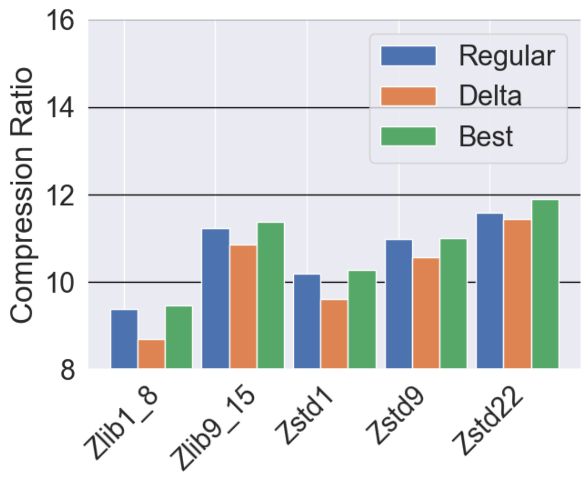
`zlib` is only able to look at 256 bytes, leading to relatively poor results. With a window size of 15, `zlib` can look at 32KB when compressing and achieve significantly larger compression ratios. Interestingly, window settings can have a larger effect than the compression level as `zlib` with a large window at level=1 outperforms `zlib` with a small window and level=5 or level=9. Therefore, we conclude that one of the most important preconditions for good compression ratio is a large window for the compression algorithm. However, this requires data to be stored in larger blocks, and also requires more memory to store larger dictionaries during compression.

4.1.2 Benefit of Delta Encoding

We consider the effect of delta encoding on the size of the compressed data. As described in Chapter 3, we perform a delta compression pass on integer data prior to compressing using the compression algorithms. We present the results of these passes for both the CMS and ATLAS experiments. We find that if used for all integer branches, delta encoding reduces overall compression ratio. However, if we create an oracle model where delta compression is only done for a branch if it provides a benefit, using delta encoding can produce a 3%



(a) CMS Data (Table 3.1)



(b) ATLAS Data (Table 3.2)

Figure 4.3: Compression Ratio With and Without Delta Encoding Pass (Integer Data)

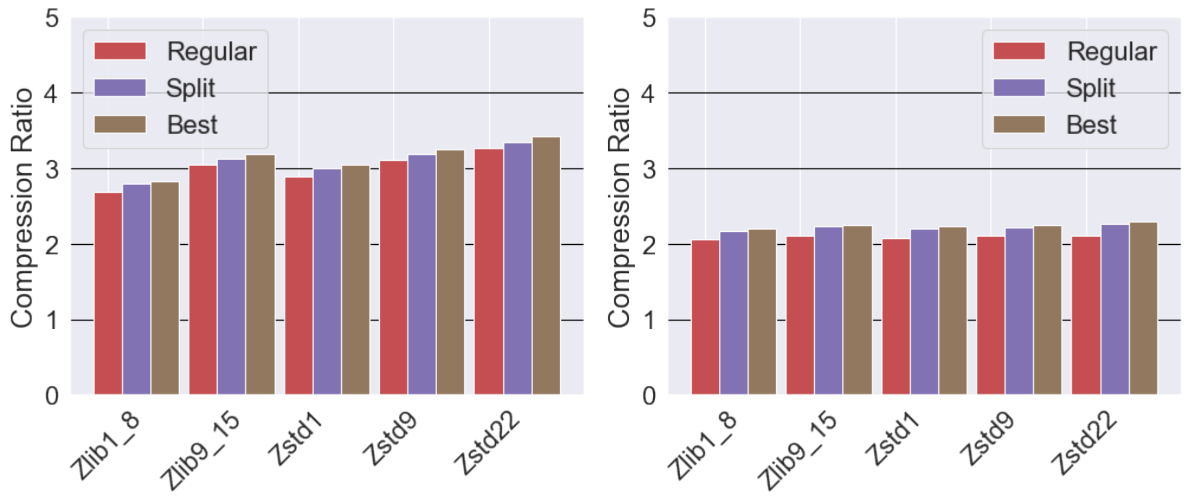
improvement in compression ratio over unmodified compression. This benefit may seem slight, but when applied to terabytes of integer data, this represents a significant reduction in data size. The oracle model of best choice does create additional computational overhead, as both options must be tested. However, the branches where delta compression provides a benefit are similar across files, and therefore could have their names stored to automatically trigger delta encoding. These branches seem to have regular strided patterns that get turned into a stream of constant or near constant values when delta encoded, providing repetition for the compression algorithm pass.

4.1.3 Benefit of Float Splitting

We also consider the effect of splitting floating point values into exponent and mantissa+sign on compression ratio. As described in Chapter 3, we perform a float-splitting pass on floating point data prior to compressing using the compression algorithms. We present the results of these passes for both the CMS and ATLAS experiments. In this case, we see that float splitting provides a benefit across the different algorithms and configurations, as shown in Figure 4.3. For `zstd` with level=9, splitting floating point values and compressing them separately increases compression ratio from 3.04 to 3.11, an 2.3% improvement over unmodified compression. Comparing between the ATLAS and CMS datasets, we observe that the floating point values from the ATLAS data have lower compression ratio. The ATLAS data contains a significantly higher proportion of double prevision floating point values, which do not benefit from the float splitting, and therefore across the floating point values the results are less significant. Although this may simply be due to greater entropy in the double precision floating point values, we conjecture that this is due to the padding that must be added to split floating point values, as described in Chapter 3.

Additionally, we find that increasing compression level does not have as significant an effect on the floating point data size for the ATLAS experiment data, with compression ratio only improving by 2% when changing from `zstd` with level=1 to `zstd` with level=22. This differs significantly from the CMS data where compression ratio improves by 11% when changing from `zstd` with level=1 to `zstd` with level=22, almost 10 times more than the ATLAS data. Although it might be that the ATLAS data contains a greater amount of entropy and is therefore just more difficult to compress, we attribute some of the discrepancy to the organization of the branches, as the ATLAS floating point branches are on average smaller. Higher algorithmic levels for compression benefit mainly from larger window sizes, so if files are small, the benefit from larger windows is nonexistent.

In similar fashion to the delta encoding experiment, we observe that if we create an oracle model where the best choice is made for every branch, splitting floating point values when it provides a benefit can improve the overall compression ratio by up to 0.1, representing an improvement in compression ratio by 5-8%. However, we recognize that across the files, the branches where float splitting does not provide a benefit are relatively constant and therefore could have their names stored to automatically bypass float splitting for them.



h]

(a) CMS Data (Table 3.1) (b) ATLAS Data (Table 3.2)

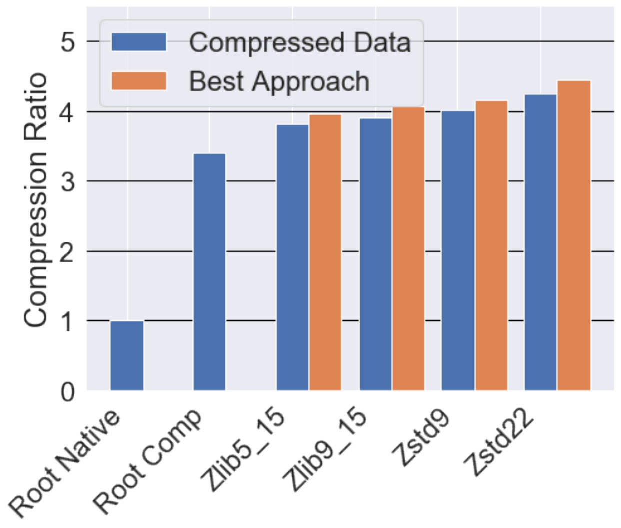
Figure 4.4: Compression Ratio With and Without Float Splitting Pass (Float Data)

4.1.4 Combined Approach

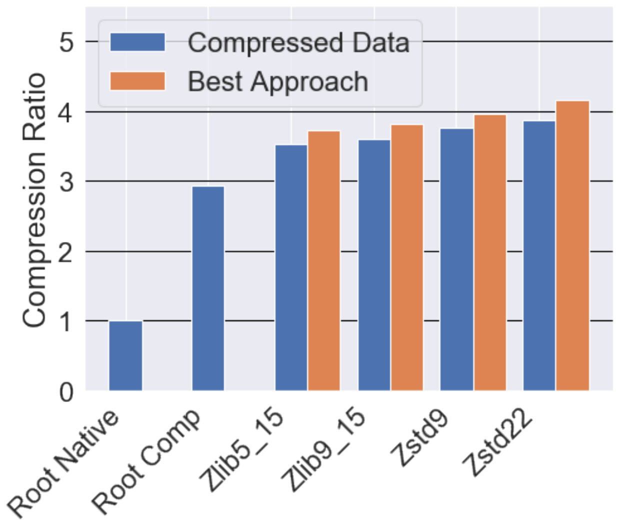
We consider the benefits offered by algorithmic choice, delta encoding and float splitting in the context of the overall data. The data distribution plays an equally important role in determining the overall benefit of the approaches. We label the combined delta encoding and float splitting approach as "Best" in Figure 4.5. We find that the combined delta encoding and float splitting approaches have a noticeable effect on the overall compression ratio when paired with `zstd`, level=22, improving compression ratio by up to 4.2% over native `zstd`, level=22. In preliminary studies, we evaluated similar techniques for boolean data, other structured data, and unsigned integer data but found that the techniques were largely ineffective at increasing compression ratio.

4.1.5 Benefit of Dictionary Usage

We consider a few methods of utilizing a pretrained dictionary to increase compression ratio on smaller files. First, we attempt to pretrain a dictionary on an entire ROOT file, and use it compress other ROOT files. We find that this approach performs very poorly, having



(a) CMS Data (Table 3.1)



(b) ATLAS Data (Table 3.2)

Figure 4.5: Best Approach (Oracle Float Split + Delta Encode) vs Original Algorithm

little effect on the compression ratio, as the bit level patterns within a file vary significantly throughout a file. Hence, a reasonably small dictionary cannot learn the patterns of a file effectively.

We also train an individual dictionary for each type of data that we compress. We create an dictionary for each size of integer, float, and unsigned integer type, as well as a dictionary for boolean and other structured types. This has a higher cost of dictionary storage, as we store 10 times as many dictionary files, but each file is relatively small and capped at under 128 KB. Hence, we see that this added cost is minimal when compared to the overall size of the file. We train each dictionary on the branches of its type from a selected file, and then evaluate the performance of the compression algorithm with a pretrained dictionary on both the original file and other files. We use the default training algorithm and settings provided by `zstd`. However, we find that in nearly all cases, this approach fails to improve compression ratio. This result is surprising, and although we do not evaluate it in this thesis, we hypothesize that the lack of similarity between branches and between the randomly sampled files hinders compression ratio. If one pretrained dictionary is used per

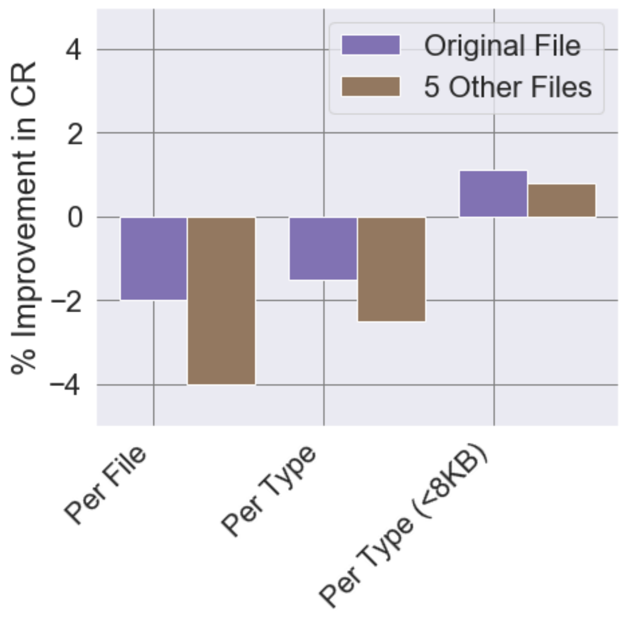


Figure 4.6: Pretrained Dictionary Approach (CMS Data, Table 3.1)

file, per dataset, perhaps the compression ratio would be improved.

We additionally consider approaches where only smaller branches (<8KB) are used for training and evaluating a dictionary, and find that with these restrictions, the compression ratio on the evaluated files can improve slightly, but not by more than 0.5% over the same compression without the dictionary. We present the results for the different training methods in Figure 4.6. These results are drawn from the CMS dataset, and represent the compression ratio on data extracted when training on the branches of 1 file, and then evaluated on 5 other files. We include the size of the trained dictionary as part of the compressed size, although this contribution could be made less costly if more files are run with the same dictionary, effectively amortizing the cost.

We can understand the poor performance of the pretrained dictionaries through an investigation of how they affect the underlying compression algorithms. By starting with a pretrained dictionary, a small file that has very similar bit level patterns to the file used for training will benefit greatly. However, for a larger file or a dissimilar file, this effect could be much weaker, as the pretrained dictionary might hinder the learning of new patterns.

Furthermore, since the bulk of the data stored in these ROOT files is in larger branches, even a significant improvement on files smaller than 8 KB would only lead to a marginal improvement overall. These dictionary approaches do have much greater benefits for compression throughput however, as the pretrained dictionaries greatly improve compression performance for `zstd` when operating at a high level.

4.1.6 Aggregated Approach

Although we find that the pretrained dictionary approach provides minimal to no benefit, we consider an aggregation approach, which could mitigate some of the challenges posed by compression performance across files. We collect all files of each experiment, and merge branches of the same name to achieve an aggregated branch of all files. We then compress these larger files, which are on average more than 20 times larger than the original files, providing larger compression windows while maintaining some sense of data regularity. We evaluate this approach on both the CMS and ATLAS data, aggregating all files and then compressing using the `zstd` algorithm. We find that this approach does have a measurable effect on performance, increasing compression ratio by up to 2.18% over data compressed identically, but without aggregation. We present the results of this study in Figure 4.7, plotting the compression ratio with aggregated files over the standard compression. This result demonstrates that aggregation of files can improve compression ratio, largely due to larger aggregate branch size improving context and history for compression algorithms.

4.2 Compression and Decompression Throughput

4.2.1 Algorithm Comparison

We begin our throughput analysis by looking at how the various algorithms, configurations, and additional passes affect the compression and decompression throughput of the data. As

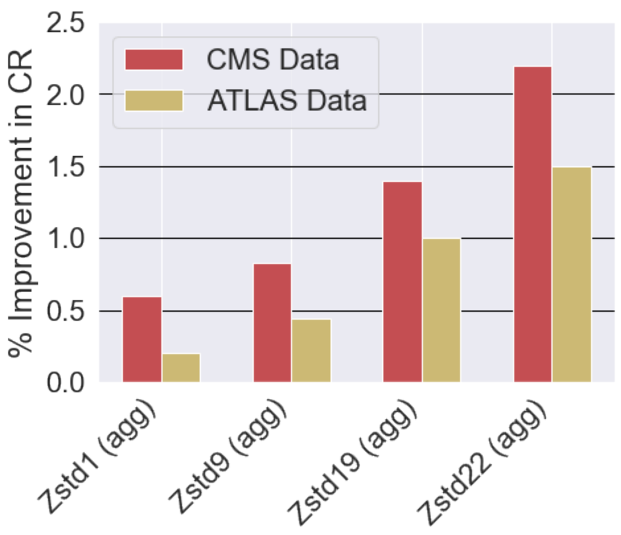


Figure 4.7: Compression Ratio Improvement on Aggregated Data

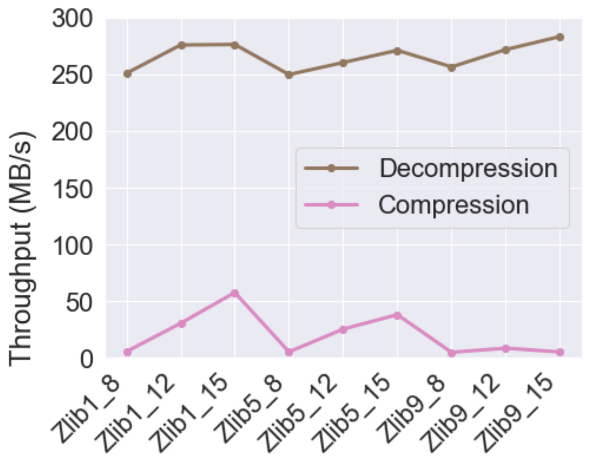
	Zlib								
	1_8	1_12	1_15	5_8	5_12	5_15	9_8	9_12	9_15
Compression	5.7	30.6	57.7	5.5	25.5	38.2	5.1	8.8	5.3
Decompression	250.7	275.6	276.1	249.6	260.2	270.9	256.3	271.6	283.0

	Zstd				Float Split	Delta Pass	Snappy
	1	9	19	22			
Compression	262.9	31.8	3.1	2.0	327.2	252.3	103.4
Decompression	673.2	736.1	709.6	716.0	327.2	252.3	300.4

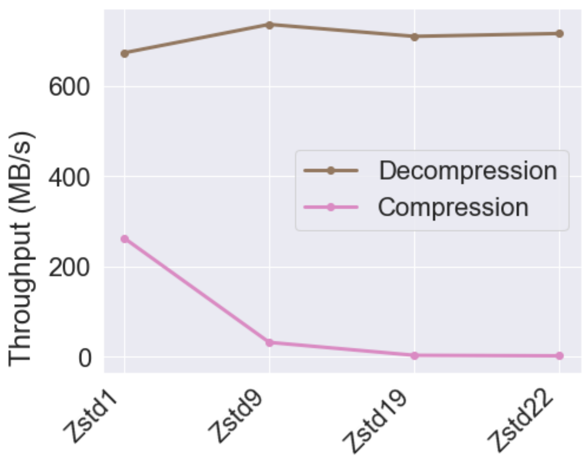
Table 4.1: Algorithm Throughput (CMS+ATLAS Data)

discussed in the metrics section of Chapter 3, these results are for single core performance. Finding a baseline for the native ROOT compression is complex, as the compression algorithms used in ROOT have throughput that varies widely based on the basket sizes and disk layout of the branches. We compare our approach against the performance of standard ROOT compression in Chapter 5.

We present the results of the compression throughput from the combined ATLAS and CMS experiments in Figure 4.8 and display the data in Table 4.1. For this analysis, we used pretrained dictionaries only for the small files, which performed best in the compression ratio experiments. We conclude that generally `zlib` compression performance increases



(a) zlib



(b) zstd

Figure 4.8: Compression and Decompression Throughput (CMS + ATLAS Data))

significantly with a larger window size as fewer context switches and compression calls are required. Additionally, as compression level increases, compression throughput drops, with compression performance ranging between 5 and 9 MB/s for level=9.

However, `zlib` performs relatively consistently for decompression performance, with all configurations performing between 250 and 290 MB/s. `zstd` also has consistently high decompression throughput, with all configurations decompression between 670 and 720 MB/s, more than twice as fast as `zlib`. However, at high compression levels we notice the compression throughput of `zstd` drops to 3 MB/s, comparable to the performance of `zlib` at high levels. Although these values seem low, we note that the primary concern for analysis workloads is the decompression speed, as it is the common use case. In Chapter 5, we show how a low throughput compression algorithm can still achieve practical data reduction for the ATLAS storage system as long as decompression performance is as similar to that of current techniques.

	Zstd Compression				Zstd Decompression			
	1	9	19	22	1	9	19	22
Standard	262.9	31.8	3.1	2.0	673.2	736.1	709.6	716.0
Dictionary	215.1	30.3	6.33	6.29	653.2	714.1	798.9	762.7
Aggregated	253.4	31.79	1.70	1.54	653.2	714.1	798.9	762.7

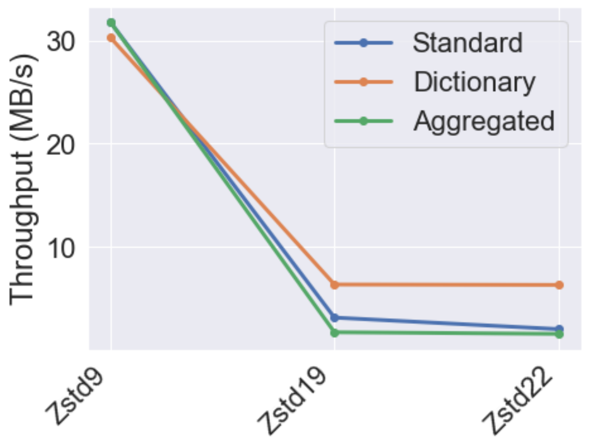
Table 4.2: Aggregated and Pretrained Dictionary Throughput (CMS Data, Table 3.2)

4.2.2 Throughput of Selected Techniques

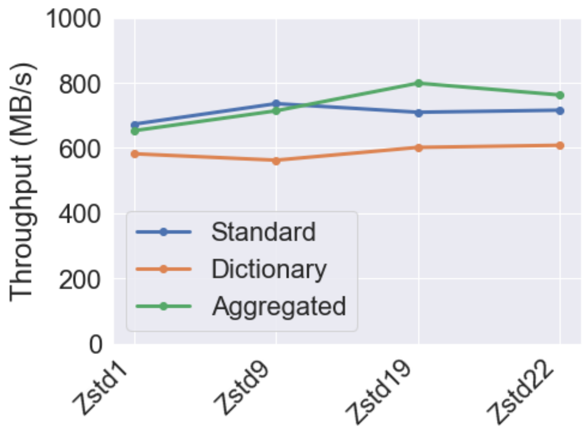
We include the throughput of the delta encoding and float splitting steps in Figure 4.8 and display the data in Table 4.1. Both passes are equivalent in terms of resource usage and computation in both directions, and we find that their performance is nearly identical over the combined datasets. The delta encoding pass performs at 250 MB/s, and the float splitting pass performs at 330 MB/s. Both provide almost no overhead on compression throughput, as they are more than 50x faster than the compression algorithms at high levels, but potentially could limit decompression throughput for `zstd`. However, we recognize that the current implementation is designed for easy verification and code readability. A production implementation of both passes could be written extremely efficiently for performance, which would likely improve the throughput of both passes significantly.

We also evaluate the performance of the aggregation and pretrained dictionary techniques to see if their use affects the performance of the `zstd` algorithm. We present the results in Table 4.2 and Figure 4.9, but we elide the result for `zstd`, level=1 from the plot for clear visualization. We find that using a pretrained dictionary significantly improves the compression throughput of `zstd`, level=22 by almost 3x, increasing it from 2.0 to 6.3 MB/s. However, this benefit is only significant at higher levels of the algorithm; at lower levels the difference is minimal, $< 5\%$ higher throughput than the original compression. Dictionary decompression is between 20 and 25% slower than standard decompression, a fairly significant cost, as decompression is the most important metric.

We find that aggregating branches across files reduces compression throughput. We notice



(a) Compression, Dictionary and Aggregated



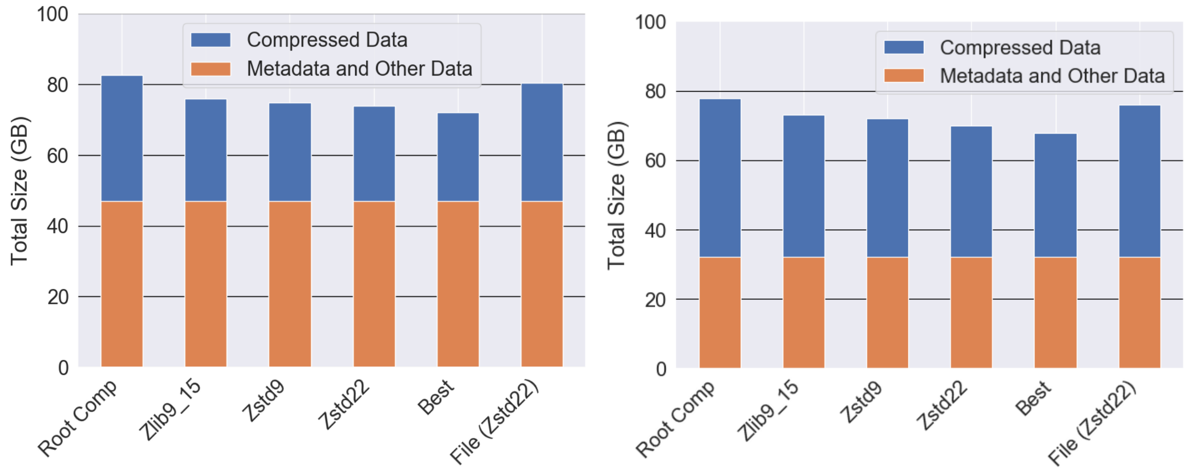
(b) Decompression, Dictionary and Aggregated

Figure 4.9: Compression and Decompression Throughput (CMS + ATLAS Data))

that above a certain file size threshold, the compression throughput of `zstd` deteriorates significantly. Specifically, we find that with `zstd`, level=22 compression throughput decreases by 25%. Hence, aggregation can benefit compression ratio, but at a cost to compression throughput. We find that the change in decompression throughput for branch aggregation is not significant across the different `zstd` levels.

4.3 Performance Summary

Fundamentally, the choice between different compression algorithms, configurations, and additional techniques is a decision about tradeoffs. To achieve a usable compression approach, we must consider approaches against goal combinations of throughput and compression ratio. Additionally, because we only compress a portion of the overall data contained in a ROOT file, we have to consider the overall effect on the resulting file size, not just the compression ratio of the extracted data to determine the storage saving potential of our approach. In Figure 4.10, we show the resulting change in the total size of the dataset vs its current compression. This represents the physical disk space that would be saved by using each approach. The `zstd`, level=22 (File) approach represents the compression ratio achieved by



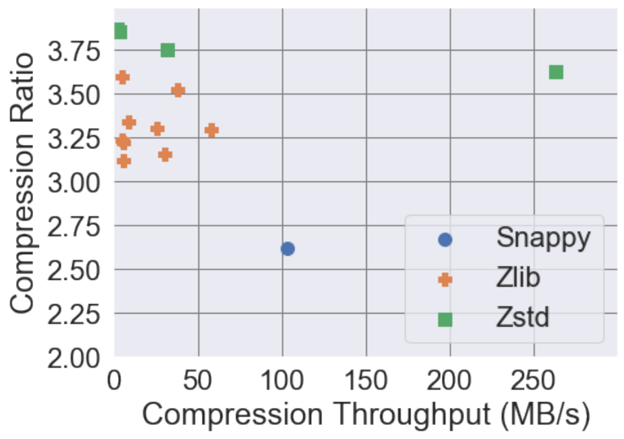
(a) ATLAS Data (Table 3.2) (b) CMS Data (Table 3.1)

Figure 4.10: Resulting File Size for Compression Approaches

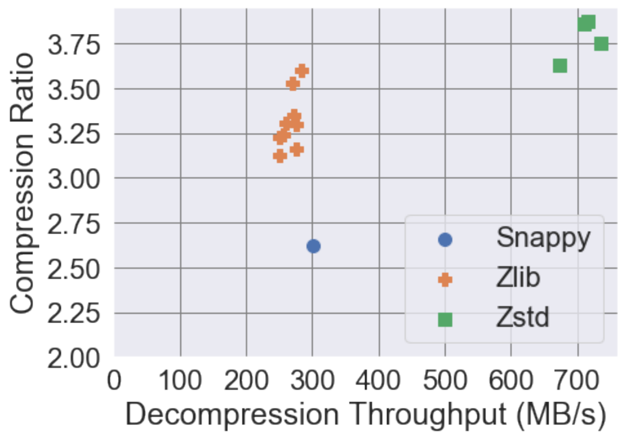
compressing the ROOT file directly, i.e., calling `./zstd 22 file.root`. We find that this only reduces the file size by about 1%. Across the different files, we find that the compression ratio is fairly constant, as the data distribution within each file is similar.

We include a comparison of running the compression algorithm with the best compression ratio, `zstd` with level=22 on the entire ROOT file. Using `zstd` with the best approach for delta encoding and float splitting improves compression ratio over the entire file by 15% for the CMS data, and 13% for the ATLAS data. These numbers represent a significant decrease in data as we can reduce the size of both datasets by about 10 GB from their original 80 GB, simply by compressing the primitive datatype.

To visualize the tradeoffs between throughput and compression ratio, we present Figure 4.11. We can see from this that `zstd` provides superior compression ratio at the cost of throughput. However, decompression throughput is the opposite, as `zstd` is significantly faster at decompressing data as compared to `zlib`. Snappy does well in throughput in both cases, but fails to provide a competitive compression ratio. However, we note that the delta encoding and float splitting passes are not a barrier to performance, and when added can improve the compression ratio by 2.8% for `zstd`, level=22. We also find that pretraining



(a) Compression



(b) Decompression

Figure 4.11: Tradeoffs Between Throughput and Compression Ratio (CMS + ATLAS Data)

dictionaries and aggregating data can increase compression throughput by up to 3x for `zstd`, level=22, as shown in Figure 4.9. The benefits are more substantial on the higher compression levels, so effectively they reduce the throughput cost of higher compression levels while providing the benefit of better compression ratio.

In the following chapter, we analyze the effect that this superior compression ratio over the native ROOT compression techniques can have on the current HEP architecture, and make recommendations on how to best integrate these compression techniques.

CHAPTER 5
IMPACT ON HEP DATA STORAGE

We see that data compression only leads to reduced storage costs if the techniques used for compression are scalable and cost effective. In this chapter, we evaluate how our approach differs from the current ROOT compression and discuss how our compression approach could be implemented in a production system. We consider how increasing basket sizes within the ROOT framework, switching to `zstd`, and using delta encoding and float splitting could improve compression ratio. Then, we provide general architectures for a scaled up compression system and model the cost of such a system for the ATLAS experiment.

5.1 Comparison Against Current ROOT Compression

At the time of the main study described in Chapter 3, `zstd` was not a supported compression algorithm for the ROOT framework. In a recent update, however, it was added to the ROOT framework, but is not yet supported by the CMS and ATLAS software packages, and therefore we cannot directly evaluate the performance of `zstd` within ROOT on the CMS and ATLAS datasets (Tables 3.2 and 3.1). However, we can make valuable estimates about the performance of `zstd` based on our evaluation in Chapter 4.

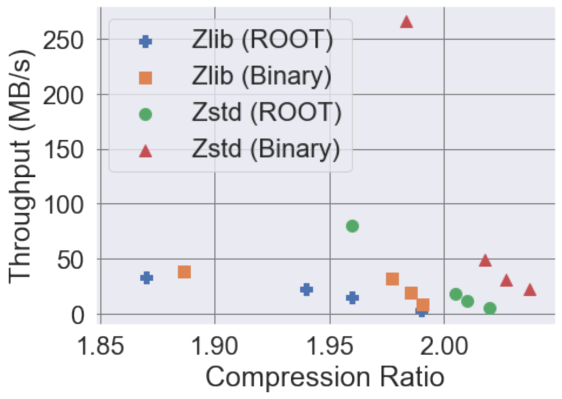
To get an understanding of how `zstd` performs in the ROOT framework, we use the techniques described in [50] and [58] and create an auto-generated 10000 event ROOT tree that contains a mix of random and patterned data. We follow a ROOT Tree example[1] to generate our tree and to evaluate the compression performance. This file is 700 MB when uncompressed, and contains a mix of integer, single, and double precision floating point values.

1. `https://root.cern.ch/root/html/tutorials/tree/tree3.C.html`

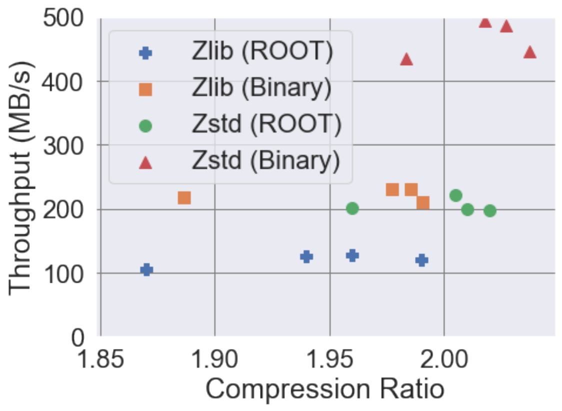
5.2 ROOT Compression Performance

For data analysis, the most important and common workloads involve reading data. Hence, we would like the read performance of compressed data to be as good or better than the currently used ROOT compression. We evaluate the read performance of data compressed within and outside of the ROOT framework, which is just a compressed binary representation of each branch's data.

5.2.1 Comparison of Throughput and Compression Ratio



(a) Compression Performance



(b) Decompression Performance

Figure 5.1: Compression within ROOT vs Compression on Extracted Binary Data

We compare the performance of using `zlib` and `zstd` both within the ROOT framework (ROOT) and and outside of the ROOT framework (Binary) where data has been extracted by branch and is stored in individual binary files. The process of extracting data was evaluated to have throughput of over 200 MB/s per core, so this process does not represent significant overhead to compressing the extracted data. In Figure 5.1, we present the compression and decompression throughput against the compression ratio of the algorithmic configurations. We see that the compression ratio and throughput of `zstd` is always better than that of `zlib` for corresponding levels. Therefore, we find that both within and outside the ROOT framework, `zstd` is preferable.

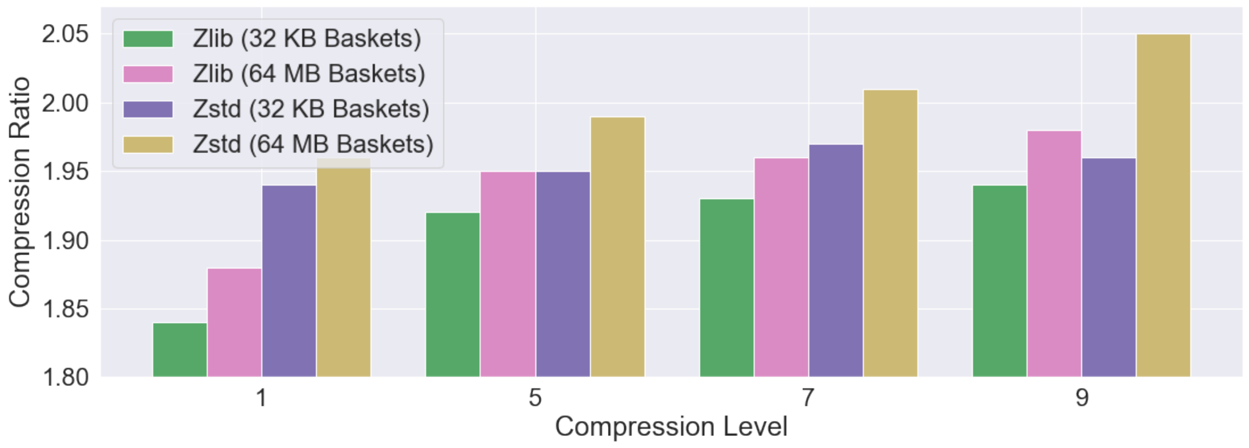


Figure 5.2: Compression Performance vs ROOT Basket Size (Random Data)

5.2.2 Basket Sizing

In Chapter 3, we found that data aggregation is an important source of compression ratio improvements,as larger blocks of data can be compressed more effectively. One important consideration when using compression within ROOT, therefore, is the size of the `TBasket`s used for compression. We use default basket sizing in the experiment presented in Figure 5.1, which ROOT sets using the average size of the first 100 events read. For this experiment, the baskets for each branch ranged in size from 5KB to 4 MB. For larger trees, the basket sizes are often quite small (32 KB), to allow baskets of many branches to be stored in memory.

Next, to determine the effect of basket sizing on compression ratio and throughput, we compare ROOT compression with small (32KB) and large (64 MB) basket sizes. We find that a basket size of 64 MB can improve compression ratio by up to 5% over the a 32 KB basket size (Figure 5.2). As we discussed in 4.1.6, the improvement in compression ratio can be attributed to the larger basket sizes filling the compression windows of the algorithms.

Even larger basket sizes could increase compression ratio more, but compressing large baskets presents a challenge, as more resources need to be acquired during the compression process. On large files with 1000 or more branches, 64 MB baskets are impractical for analysis as they require more than 64 GB of RAM just to store one basket of each branch.

We conclude that these basket sizes are practical for data center compression, where machines can be expected to have sufficient memory to handle larger basket sizes. However, supplying files to clients with these large basket sizes is not feasible, as we do not expect users to have more than 2GB of RAM/core. Hence, if we wish to obtain the benefit of large baskets for compression, we must use strategies which rewrite the data in a client-readable way for analysis.

5.3 Modeling a Production Implementation

From our study, we find that within the ROOT framework, `zstd` levels 7 and 9 delivers better compression ratio and throughput performance than `zlib` levels 7 9, respectively. Furthermore, based on the analysis in Chapter 4, adding support for `zstd` levels beyond 9 to the ROOT library would enable a 3-4% improvement in compression ratio over `zstd`, level=9. From our experiments, we know how well a variety of techniques and algorithms perform on the CMS and ATLAS data, and now we can use them to design a infrastructure which reduces the data storage requirements of the ATLAS storage system.

We consider the computational cost and storage cost of our strategies. We note engineering costs where applicable, but do not quantitatively measure them. To understand and compare the cost of these infrastructures, we define a set of metrics.

Infrastructure Metrics:

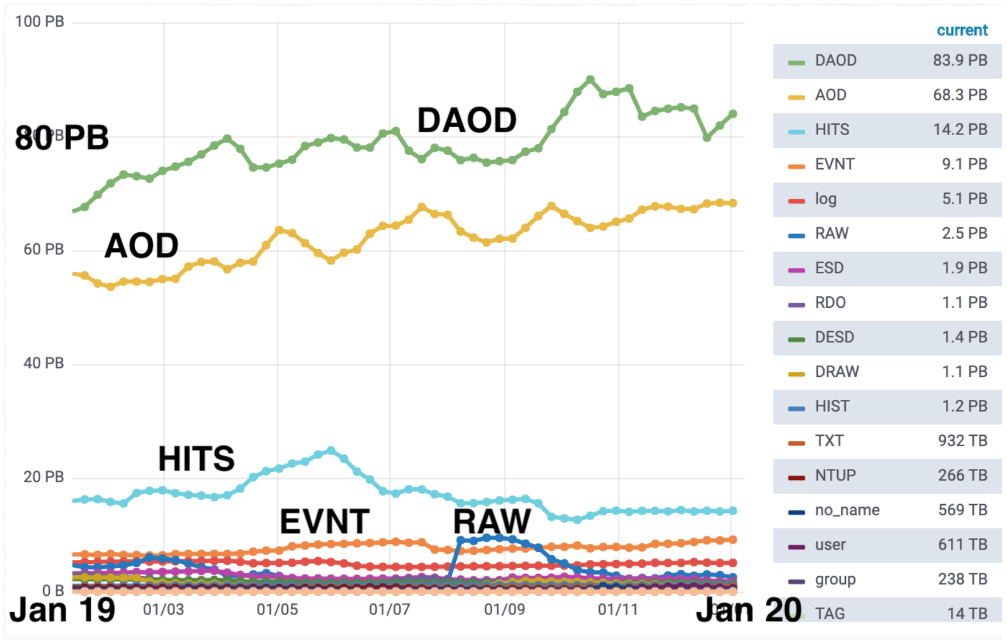
- Core-Hour: The amount of work done in one hour by a single core on the evaluation system (2.83 GHz Xeon E5440, as described in Section 3.6). The CERN Tier0 grid has 224 thousand cores, which represents a computation capacity of approximately two billion core-hours per year [15].
- Read Cost: The number of server-side core-hours needed to decompress and provide 1 petabyte of data from a compressed representation. We consider the current Read

Cost of the ATLAS storage system to be 0 core-hours/PB, as files can be provided to the client without modification.

- Write Cost: The number of server-side core-hours needed to compress and store 1 petabyte of data from a uncompressed representation. The current Write Cost of the ATLAS data storage system for `zlib`, level=5 is 10,900 core-hours/PB.

- Storage Reduction: The size of storage space saved by a given strategy, measured in petabytes.

We evaluate the storage savings against the DAOD data stored by the ATLAS experiment. This data is currently stored with `zlib`, and comprises 85 PB of data (Figure 5.3). From data provided by the CERN IT dashboard, we estimate the read-write balance of ATLAS data at 96/4, meaning that for every petabyte of data written to the ATLAS system, approximately 24 petabytes are read [15].



	current
DAOD	83.9 PB
AOD	68.3 PB
HITS	14.2 PB
EVNT	9.1 PB
log	5.1 PB
RAW	2.5 PB
ESD	1.9 PB
RDO	1.1 PB
DESD	1.4 PB
DRAW	1.1 PB
HIST	1.2 PB
TXT	932 TB
NTUP	266 TB
no_name	569 TB
user	611 TB
group	238 TB
TAG	14 TB

Figure 5.3: Current ATLAS Storage Usage

We use the combination of ATLAS data surveyed and the auto generated file to model the performance of `zstd` with different levels, basket sizes, delta encoding, and float splitting, on the current set of ATLAS data, which we show in Figure 5.3. The representative sample of ATLAS files we studied in Chapter 4 are a portion of the 85 PB of DAOD data stored for the ATLAS experiment, so we can use it to predict the performance of our strategies on the entire dataset. We use [15] to estimate the data written to ATLAS DAOD in 2019 to be 15 petabytes, and the data read to be approximately 350 petabytes.

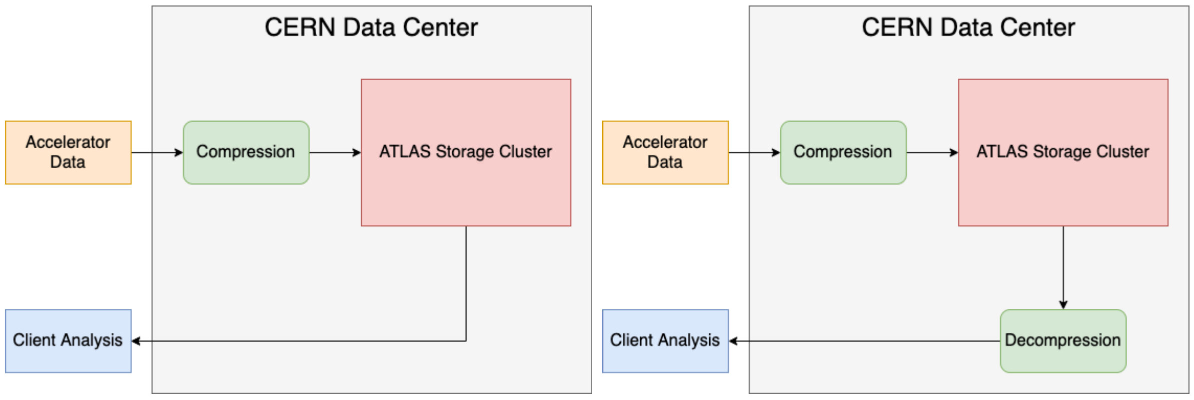
5.4 Proposed Integrations

We use our data from Chapter 4 to design compression strategies for production use. They vary in implementation cost, compression ratio, and computational cost to provide a comprehensive set of options. These strategies are designed to have better compression ratio than the currently used `zlib`, level=5, equal or better decompression performance, and a range of compression throughputs. Generally, we find better compression ratio corresponds with lower compression throughput. In a production environment, this tradeoff and the underlying workload would inform the strategy chosen, and its specific configurations. We describe these strategies in detail, estimate their core-hour cost for the 2019 reading and writing patterns, and estimate the storage reduction for the ATLAS experiment.

5.4.1 Strategies

1. ROOT `zstd`: The simplest change to the current compression strategy is to switch compression algorithms to `zstd`, level=7 or 9, within ROOT. We found that `zstd`, level=7 outperforms the current compression used by the ATLAS experiment (`zlib`, level=5) in both compression ratio and throughput. `zstd`, level=9 provides even better compression ratio than level=7, but at a small cost to compression throughput.

2. ROOT `zstd` + DE/FS: This strategy switches to `zstd` level=7 or 9, but also integrates the delta encoding and float splitting passes into the ROOT system. This would



(a) Strategies 1 and 2 (b) Strategies 3, 4, and 5

Figure 5.4: Dataflow Models

require a software engineering effort to add both passes into the ROOT library branch streamers. This strategy benefits from requiring no server side decompression, but exploits the benefits that the delta encoding and float splitting passes provide.

3. ROOT `zstd` + DE/FS + Basket Resize: This strategy switches to `zstd` level=7 or 9 within ROOT, integrates the delta encoding and float splitting, and increases the size of baskets during storage. Client analysis would be done on uncompressed data with this strategy.

4. Binary `zstd`: This strategy stores compressed binary representations of all branches with `zstd` level=19 or 22, and uses delta encoding and float splitting. Client analysis would be done on uncompressed data with this strategy.

5. Binary `zstd` + Pretraining : This strategy stores compressed binary representations of all branches with `zstd` level=19 or 22, uses delta encoding and float splitting, and pretrains dictionaries for small files. Client analysis would be done on uncompressed data with this strategy.

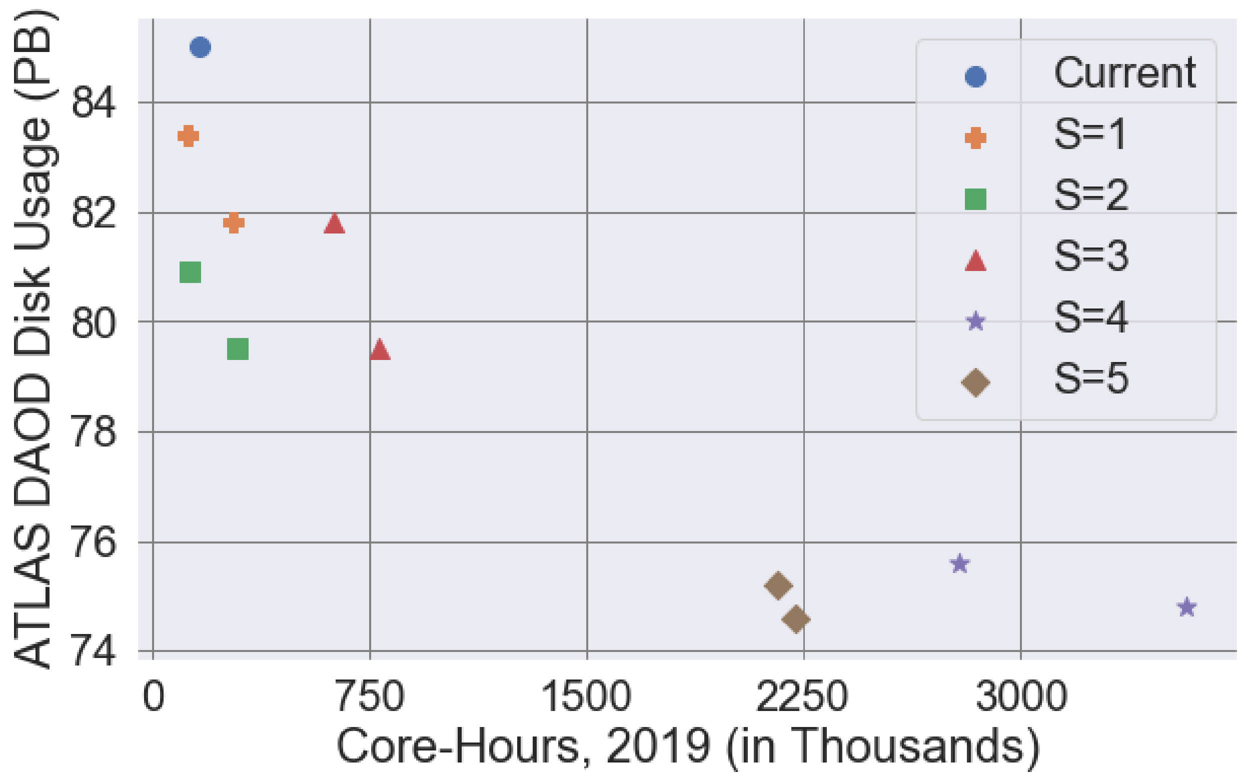


Figure 5.5: Comparison of Strategies and Core-Hours for 2019 Usage

5.4.2 Analysis of Strategies

We compute the estimated core-hours required to adopt each strategy for the ATLAS system based on the data transfer statistics from the 2019 year. We note that these values are computed with the assumption that all data is compressed with the selected strategy, but in a production environment, more frequently accessed files could be cached in an uncompressed format to speed up access times and reduce decompression overhead. While these calculations are only for ATLAS DAOD, if we scale the data volume to match the capacity of the entire CERN grid (400PB), we see that even the most computationally expensive approach (S=4), represents less than 1% of the total core-hours used by the CERN grid.

1. Switching the compression algorithm used internally by ROOT offers significant gains to both compression ratio and decompression throughput. With `zstd`, level=7, compression ratio shows a 2% improvement over `zlib`, level=5 and a 40% increase in com-

Strategy #	Zstd Level	CR over Zlib5	Read Core-Hours (1000s)	Write Core-Hours (1000s)	Storage Savings (PB)
0	-	1.00	0	164	0
1	7	1.02	0	117	1.7
1	9	1.04	0	276	3.3
2	7	1.05	0	128	4.0
2	9	1.07	0	288	5.6
3	7	1.04	456	168	3.3
3	9	1.07	456	326	5.6
4	19	1.12	1400	1390	9.1
4	22	1.14	1400	2173	10.4
5	19	1.13	1500	661	9.8
5	22	1.15	1500	725	11.1

Table 5.1: Comparison of Production Strategies for 2019 Usage

pression throughput. With `zstd`, level=9, compression ratio shows a 4% improvement over `zlib`, level=5, but a 40% decrease in compression throughput. Both strategies increase decompression throughput by 2.6x . As shown in Figure 5.5 and Table 5.1, this strategy, displayed as S=1, could reduce data storage requirements by up to 1.7 PB while providing better read/write performance to an end user than the current `zlib`, level=5.

2. We observe that using delta encoding and float splitting with the `zstd` algorithm in ROOT can provide significant compression ratio benefits over strategy 1 with minimal cost to read/write performance. With `zstd`, level=7, compression ratio shows a 4% improvement over `zlib`, level=5, equivalent decompression throughput, and a 35% increase in compression throughput. With `zstd`, level=9, compression ratio shows a 7% improvement over `zlib`, level=5, equivalent decompression throughput, but a 45% decrease in compression throughput. As shown in Figure 5.5 and Table 5.1, this strategy, displayed as S=2, could reduce data storage requirements by 5.6 PB with fewer than 20,000 core-hours needed per PB of data added. Additionally, for both this strategy and strategy 1, no decompression or processing is needed before data can be

used, so the read core-hours/PB is 0.

3. Unlike strategies 1 and 2, strategy 3 requires decompression at the server side (Figure 5.4) before clients can run analysis on the data. Because storing data with very large baskets can present challenges for ROOT analysis, we decompress the data when providing it to a client. This uses only 1300 core-hours/petabyte. This strategy shows an 4-7% improvement in compression ratio relative to `zlib`, level=5, and can save between 3.3 and 5.6 PB of data depending on the level of `zstd` chosen.

4. Strategy 4 extracts branches prior to storage, reducing the computational overhead of operating within the ROOT framework. We find that this strategy can provide a 12-14% compression ratio improvement over `zlib` with level=5. This corresponds to a savings of 9.1 to 10.4 PB of disk space for the ATLAS DAOD experiment data. As shown in Figure 5.5 and Table 5.1, this strategy, displayed as S=4, requires significantly more read and write core-hours per PB than the previous strategies. Additionally, this strategy incurs some engineering overhead, as high performance automated software to extract data from ROOT data files has to be written.

5. Finally, strategy 5 offers the best reduction in storage utilization, reducing data storage requirements for the ATLAS DAOD data up to 11.1 PB. This represents a 13% reduction in data storage requirements. This strategy is less computationally expensive for compression than strategy 4, as the pretrained dictionaries reduce the core-hours by almost 70% for the level=22 case, and 50% in the level=19 case. However, the read core-hours are similar to strategy 4, as the dictionary usage does not increase decompression throughput. Hence, we see that this strategy provides the best compression ratio, but at a higher read cost than the first 3 strategies.

5.4.3 General Discussion

The system architecture for the strategies is displayed in Figure 5.4. Strategies 1 and 2 require no additional components, as they simply modify the compression block for data added. However, for strategies 3, 4, and 5, decompression or data reorganization has to be done before data can be used for client analysis.

In these calculations, we assume that data analysis can operate at equal or better performance with uncompressed ROOT files. However, if certain analyses require compressed data for backwards compatibility, this could be done with additional cost. We find that recompressing data with `zstd`, level=7 would increase the core-hours for strategies 3, 4, and 5 by 2.6 million. Still, the most computationally intensive strategy would only use 1.5% of the CERN datacenter computation. On the other hand, we point out that many analysis workloads may not require the ROOT framework, and therefore providing data without repacking it into ROOT files could reduce the read cost for strategies 4 and 5 by almost 50%. Strategies 4 and 5 would also provide the ability for a client to request specific tree or branch from a file or set of files, speeding up analysis drastically if only a small portion of data from a file is needed.

We also note that a couple strategies not modeled in this section could prove useful for certain scenarios. For larger data collections (consisting of many files often accessed together) file-level aggregation could provide a way to increase compression ratio with `zstd`, level=22. However, if one file needs to be accessed, the whole collection must be decompressed as there is currently no way to decompress only part of a compressed file with `zstd`.

Overall, we conclude that all of the 5 strategies significantly reduce the data storage requirements of the ATLAS experiment. They range from simple to complex in terms of implementation, and have savings that track accordingly. It is clear that ATLAS should use `zstd` for data storage as soon as it is supported by all necessary software packages. We find that strategy 5 offers an intriguing possibility for future storage design, as the cost to read

Strategy #	Zstd Level	Storage Savings (PB)	Total Core-Hours (1000s)	Storage Savings (1000s of $)	CPU Cost (1000s of $)	Net Cost (1000s of $)
0	-	0	164	0	5.6	-10
1	7	1.7	117	170	4.0	170
1	9	3.3	276	330	9.4	320
2	7	4.0	128	400	4.4	400
2	9	5.6	288	560	9.8	550
3	7	3.3	624	330	21.2	310
3	9	5.6	782	560	26.6	530
4	19	9.1	2790	910	94.9	820
4	22	10.4	3573	1040	121.5	920
5	19	9.8	2161	980	73.5	910
5	22	11.1	2225	1110	75.7	1030

Table 5.2: Cost of Production Strategies for 2019 Usage

data is still fairly low and the write cost is fairly low. Although quantifying dollar costs is complex for a system like ATLAS, we attempt to do so in the following section.

5.5 Cost Estimate

To properly analyze the tradeoffs and cost of implementing our compression strategies, we need a cost model to determine how the increase in computation relates to the decrease in storage costs. We model the cost of a CPU core for an hour at \$0.034, as found on AWS and Google Cloud [8] [28]. We also use AWS to estimate the storage cost of a petabyte to be about \$100,000 per year for mid tier storage [9]. This is a fair comparison, as both values represent the amortized cost of the devices, and include the overhead of a large datacenter in their costs. These costs may be different than the actual CERN datacenter costs, but provide some way of evaluating the cost benefit of our strategies.

We first compute the cost of our strategies for the 2019 usage data based on the data in Table 5.1 and display the total savings in Table 5.2. We compute the CPU cost assuming no current CERN cores have available tune. This is likely an overestimate, as CERNs CPU utilization is not 100%, and therefore compression core usage could be shared with existing workloads. Using strategy 5 could have saved more than \$ 1 million dollars in 2019 by reducing the data storage requirements by 11.1 PB but only requiring 2 million core-hours (<0.1% of total CERN core-hours).

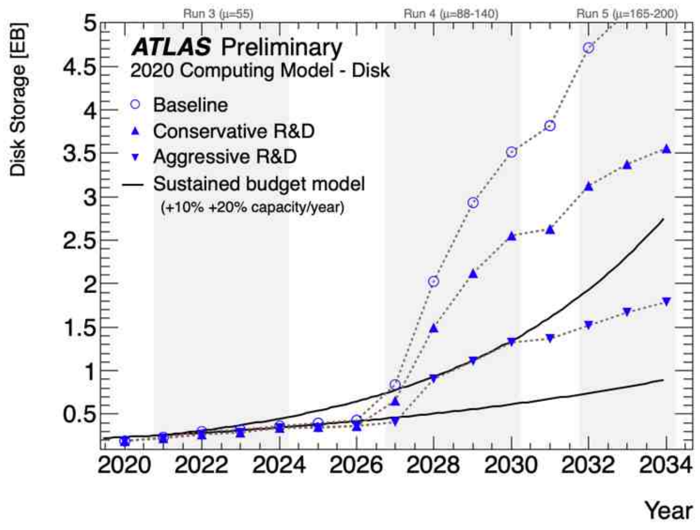


Figure 5.6: Projected HEP Data Storage Usage and Cost

Source: CERN Twiki - AtlasPublic Computing and Software

We also predict the performance of our strategy for future years, as data storage requirements are expected to grow significantly. We use the ATLAS data projections in Figure 5.6 as a rough estimate of data storage requirements. We estimate the compression data reduction and costs of our approaches for 2021 to 2034 on DAOD data. We assume the general data distribution is the same as 2020, and that data read-write balance remains the same. We use the baseline budget model curve to generate our results as it is a realistic estimate of the total data volume. We display these savings per year in Figure 5.7, where we use the higher `zstd` level for each strategy. We compute that with strategy 2, the estimated cost savings for 2034 are \$8 million, and for strategy 5, we can predict a savings of \$26 million.

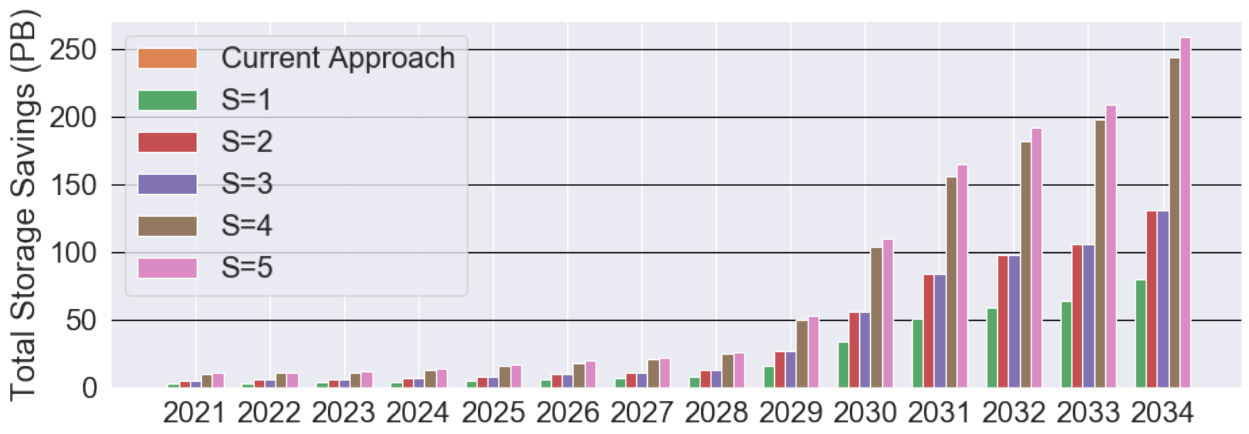


Figure 5.7: Projected ATLAS Data Reduction

5.6 Summary

Overall, we find that the benefits of storage focused data compression for the ATLAS experiment vastly outweigh the costs. Although the large scale implementation of data extraction and aggregation is no small feat, other simpler strategies such as switching to `zstd` for all data can have significant compression benefits and save petabytes of data (and therefore millions of dollars) over the next 10 years. Furthermore, with hardware acceleration approaches and custom compression techniques for other object types, the compression ratio benefits will likely increase over the next few years. As Figure 5.6 demonstrates, the ATLAS data storage requirements will increase rapidly with LHC Runs and 4, making these approaches invaluable. Across the entire high energy physics data storage infrastructure, storing data in compressing oriented formats, and using algorithms designed for high compression ratio when storing data has the opportunity to save many petabytes of data, while requiring minimal computational overhead.

CHAPTER 6
RELATED WORK

Data compression has been a central focus for high energy physics and the scientific community at large for many years. Much of the work on compression focused primarily on the effect of changing compression settings and adding additional compression algorithms and configurations to ROOT. These approaches treat the compression algorithms as a sort of black box, and do a parameter search to optimize for a particular case. In contrast, our work looks at the compression algorithms to understand how to best supply data to them. Then, we consider how we can structure data both within and outside of ROOT to best improve compression ratio.

There is also a substantial body of work on type-specific compression techniques, which range from integer and floating point specific compression libraries to complex delta object storage. These approaches often have a tradeoff between compression ratio and applicability. A heavily tailored approach for one setting might achieve fantastic results, but be unusable for more general data storage. However, our design makes it simple for any of these approaches to be easily integrated as part of a compression pipeline for ROOT storage.

6.1 Data Storage Reduction for High Energy Physics

6.1.1 ROOT Compression Optimization

As ROOT has expanded as a software toolkit, compression algorithms have been added to allow for more flexibility and customization. These additions have been influenced by a variety of research work which explored the tradeoffs that new compression libraries provide for compressing high energy physics data.

One such optimization, proposed by [13], converted objected level streamers into member streamers, allowing reads to a member of an object to occur faster and with fewer disk reads and writes. Work by Zhang et al. also studied the effects that compression algorithms can have on the size and performance of ROOT files [58]. In addition to internal algorithmic choices, they evaluated filesystem level `zlib` compression of ROOT files, and saw a degradation in compression ratio as compared to native ROOT `zlib` compression. However, their analysis did not consider the potential for storing the data outside of the ROOT file structure. [50] analyzes the various different flavors of LZ compression and their effects on throughput for data from a variety of high energy physics experimentsis Other work by the ATLAS collaboration, investigates the best ROOT framework settings for 2 different goals: read throughput and storage size. Their results show that on a ATLAS workload, `zlib` provides a 10% slower Event rate (rate at which events can be read from disk) than `lz4`, and more than 15% larger file sizes than `lzma` for the same compression level. They note that `lzma` is better for storage size minimization and that `lz4` is better for read performance, but leave an analysis of how to adaptively choose between them as future work. Their work proposes the use of `zlib` as an intermediate to enable good performance in all cases, as it represents a middle ground within the current ROOT framework.

In contrast, our work argues that this intermediate approach is costly to data storage goals, and converting data to a storage oriented format can save petabytes of data and deliver similar read performance. We evaluated `lzma` in preliminary studies but found its compression ratio benefits to be similar to that of `zstd`, but with lower throughput for both compression and decompression. On the other hand, we found `lz4` to provide better compression throughput at low levels, but it could not provide a competitive compression ratio to `zstd` to make it worth using for long term storage.

This work also investigates the configuration of ROOT cache sizes, frequency of disk writes, and multithreaded performance for ATLAS file reads. These configurations appear to be workload and dataset dependent, but are independent of our compression approach. We focus on providing competitive performance for reading and writing with current ROOT compression techniques while saving storage space through a variety of compression techniques which look at dictionary training, aggregation, and data representation.

6.1.2 Lossy Compression

Although lossy compression is not the focus of this work, there is a large body of work on the potential for lossy compression in high energy physics. [14] argues that scientific experiments must use more lossy compression to keep up with ever growing data and limited RAM and computational power. When the precision of the datatype is higher than the actual precision of the scientific instrument used to make the measurement, the precision is meaningless and can be discarded when compressing. Ongoing work at Argonne National Lab by Mete et al. focuses on masking out these "noise" bits to provide better compression. If a single precision floating point value is stored with the precision of 7 decimal digits but the instrument used to measure them can only measure up to 3 decimal digits, the last 4 digits are not actual data. Hence, we can mask out the least significant bits in the mantissa to remove the corresponding bits. This approach could be combined with our float splitting technique to improve compression ratio even more.

Lossy compression is one of the most promising uses for autoencoders and deep learning, and [30] explored the performance and usability of deep autoencoders for compressing high energy physics data, with reconstruction quality good enough for certain analysis applications. Ongoing work in this area is focused on event based and branch based predictive modeling through a variety of deep learning techniques. However, lossy approaches necessarily lose information and would require the acceptance of scientists to have significant impact. This compression cannot be undone, so any future analysis would only be able to use the reduced data.

6.1.3 Filtering at Data Source

The simplest and most substantial means of reducing data storage requirements for high energy physics analysis is through real time event selection at the data source. ATLAS and CMS both use a combined hardware and software “trigger” to process data [35] [3]. This process selects less than 0.1% of data as relevant for analysis, drastically cutting down on data transfer and storage requirements. Custom hardware approaches could be used to implement effective compression cheaply. On a large portion of the high energy physics data, even fast compression approaches like Snappy cut down data size significantly ($> 50\%$) and have been implemented on high throughput accelerators [26]. These approaches are being used to cut down the data size that has to be transmitted from the Level 1 trigger.

Additional related work by CERN focuses on the actual storage mechanisms used for archive. This body of work evaluates the tradeoffs of different storage architectures, redundancy plans, and models the cost and performance of various systems. In [24], the frequency of rewriting for reliable tape storage and the relative distribution of data access are discussed and modeled for the future. Other recent works by CERN include [40], which discusses the systems used to evaluate the distributed storage clusters used by CERN, [21], which details the process of transition to the CERN Tape Archive, and [47], which discusses the transition from CASTOR to EOS, and the future of the CERN fileystem. These works inform our analysis and provide context to discuss the costs and benefits of our compression techniques.

6.2 Data Specific Compression

A variety of domain specific techniques for data compression have emerged in recent years as general purpose compression improvements have not kept pace with the increase in data storage. We discuss works in three different areas here and compare techniques to similar ones that we have analyzed in this work.

6.2.1 Type Specific Compression

Both primitive types and more complex object types have inherent structure that can be used to compress them more efficiently. General purpose compressed cache architectures, as studied in [11], use heuristics to predict datatypes and apply techniques based on the predicted type. These techniques rely on the same underlying insight as our approach: no one compression technique performs best across the board. Another approach to a compressed object store, presented in [53], uses a form of delta compression to create a difference between a base object and only store the differences when multiple objects of the same type are used. This approach could be integrated into our work for ROOT data, but would require the implementation of automated parsing to choose base types for each object stored.

Floating point data makes up a significant portion of many scientific datasets, and correspondingly there have been a variety of floating point specific compression algorithms, designed both for throughput and compression ratio [44] [38]. These approaches use the specific design of the floating point specification to predict the bit level patterns of future values based on previous values. Other datatypes such as integers and strings have been studied with similar type-specific results. [48] uses SIMD instruction sets to enable efficient null suppression and Elias gamma encoding for integer data. For string data, work in [10] demonstrated that a tokenization style technique could be used to effectively compress string data.

6.2.2 Scientific Data Compression

Scientific computations that involve data movement are often limited by memory bandwidth, making data compression a central focus to improve application performance. In earlier work, we studied one such application, sparse matrix vector product, and found that similar data compression techniques such as delta encoding can be used to increase computational performance by more than 2x [45]. Other approaches use domain specific information to improve compression performance for both computational and storage purposes. Climate data exhibits tremendous spatial and temporal similarities, which [31] used to design a compression algorithm that achieves compression ratios better than other floating point compression algorithms. [55] analyzes power grid data, which has inherent patterns of activity and inactivity that can be used to develop better compression schemes.

6.2.3 Pretrained Dictionary Compression

Dictionary coders are used in many lossless compression algorithms, and their success depends on the established dictionary used to represent the original data. The use of a static dictionary to aid in compression is an idea often used in text compression which has a fixed set of bit patterns [37]. However, online approaches that utilize a scanning window to modify a changing dictionary are often more effective when data patterns change. LZ77 and LZ78 both use forms of online dictionary coding to achieve high compression [60] [59]. Taking ideas from deep learning, many modern approaches to image compression use pretrained dictionaries to aid in finding patterns for smaller files [17]. These techniques use a similar approach to Zstandard and its use of pretrained dictionaries to increase the compression ratio of small files. In our approach, this enables high compression ratio for smaller branches where sufficient data to fill an algorithm's window is not available.

6.3 Summary

As data storage requirements for high energy physics continue to grow rapidly, new techniques are necessary both to pare down the data being stored and to compress data that must be stored. Although lossy compression is acceptable when data precision is not essential, lossless compression continues to be used in the vast majority of cases. As the compression ratio improvements of general purpose compression plateau, data specific techniques are becoming increasingly popular and can achieve significantly better compression ratios at often low resource utilization. We see that the general trend of this work in data storage for scientific data is towards the use of a combination of high performance compression algorithms and data specific techniques to minimize storage costs for long term storage.

CHAPTER 7
SUMMARY AND FUTURE WORK

7.1 Summary

Data compression is not a black box: information about the data and its properties should be used to design techniques to better compress data. These techniques can be used to improve the currently used compression approaches in high energy physics data storage.

We study the current architecture of high energy physics data storage and find that even though data is being compressed, the format used is better suited for computation and therefore provides a relatively poor compression ratio. We evaluate the widely used algorithms `zstd`, `snappy`, and `zlib` to understand how their settings and configurations affect the throughput and compression ratio of high energy physics data. Then, we develop and evaluate a number of additional techniques that have the potential to increase either throughput or compression ratio by increasing both the repetitiveness of data and by improving the compression algorithms ability to extract information. These approaches can be broadly characterized by two similar motivations. First, we use delta encoding, float splitting, and aggregation of branches to increase the potential repetition of data to allow for compression algorithms to find more occurrences of bit level patterns. Secondly, we use larger compression windows, pretrained dictionaries, and aggregation across files to provide better context compression algorithms. These approaches deliver a better compression ratio at a various computational costs.

Evaluation on real datasets from the ATLAS and CMS experiments shows that adopting algorithms designed for modern processors and larger memory sizes can provide compression ratio improvements of 7% while providing better compression and decompression throughput. Furthermore, we describe and evaluate techniques that take into account the underlying type of a block of data can increase compression ratio by an additional 5%. Overall, we find that our approach can reduce the overall size of data files by more than 15%, providing a significant reduction in data storage requirements. We find that it is cost-effective to implement these solutions, and could reduce ATLAS data storage requirements by more than 8 PB. Our results are practical, and indicate that in high energy physics and in other domains, integrating data insights into compression techniques can drastically reduce storage requirements.

7.2 Future Work

This study can be extended in a few different ways both to more widely implement our techniques and to study other related compression strategies.

First, we only focused on compressing basic datatypes and some simple structured objects. The techniques that we applied, such as delta encoding and float splitting, can be extended to work for structured objects using more complex differencing or splitting function. These differences could be done using subtraction, exclusive or, or other reversible bitwise operations.

Secondly, future work could explore the use of auto-encoder neural networks to effectively predict the patterns of data within a branch. This approach has been developed for use in lossy compression, but has yet to be implemented for a lossless approach. This technique could be combined with a delta encoding or compression algorithm to compress the deltas between the actual data and predicted data.

We evaluated the use of pretrained dictionaries, but with much larger datasets, we predict that there are much better approaches for training and using dictionaries for compression. There are open questions surrounding what data is useful for training, how many dictionaries should be stored, and whether these dictionaries can be used across files or experiments.

Finally, it would be interesting to consider the potential of compression accelerators such as [45] and [25], to mitigate the high computational overhead of compression. At the scale of high energy physics, the computational load to compress petabytes of data becomes extremely costly both from energy and hardware. FPGA and ASIC based accelerators could enable faster compression throughput while maintaining equally good compression ratio.

REFERENCES

[1] 7-zip. Igor Pavlov. https://www.7-zip.org/.

[2] lz4, Mar 2020. https://github.com/lz4/lz4.

[3] Morad Aaboud, Georges Aad, Brad Abbott, Jalal Abdallah, O Abdinov, Baptiste Abeloos, Rosemarie Aben, OS AbouZeid, Nadine L Abraham, Halina Abramowicz, et al. Performance of the atlas trigger system in 2015. *The European Physical Journal C*, 77(5):317, 2017.

[4] Georges Aad, B Abbott, J Abdallah, AA Abdelalim, A Abdesselam, O Abdinov, B Abi, M Abolins, H Abramowicz, H Abreu, et al. Performance of the atlas trigger system in 2010. *The European Physical Journal C*, 72(1):1849, 2012.

[5] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[6] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. *ACM Trans. Inf. Syst.*, 37(1), December 2018.

[7] Johannes Albrecht, Antonio Augusto Alves, Guilherme Amadio, Giuseppe Andronico, Nguyen Anh-Ky, Laurent Aphecetche, John Apostolakis, Makoto Asai, Luca Atzori, Marian Babik, et al. A roadmap for hep software and computing r&d for the 2020s. *Computing and Software for Big Science*, 3(1):7, 2019.

[8] Amazon. Ec2 pricing, 2020. https://aws.amazon.com/ec2/pricing/.

[9] Amazon. S3 pricing, 2020. https://aws.amazon.com/s3/pricing/.

[10] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving string compression. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 655–663, 1996.

[11] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 38–49, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[13] Philippe Canal, Brian Bockelman, and Rene Brun. Root i/o: The fast and furious. *Journal of Physics: Conference Series*, 331:042005, 12 2011.

[14] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[15] CERN. Cern it overview - grafana, 2020. https://monit-grafana-open.cern.ch/.

[16] CERN. Key facts and figures – cern data centre, 2020.

[17] Yefei Chen and Jianbo Su. Sparse embedded dictionary learning on face recognition. *Pattern Recognition*, 64:51–59, 2017.

[18] CMS collaboration et al. The cms trigger system. *arXiv preprint arXiv:1609.02366*, 2016.

[19] Yann Collet and Murray S. Kucherawy. Zstandard compression and the application/zstd media type. *RFC*, 8478:1–54, 2018.

[20] Kenny Daily, Paul Rigor, Scott Christley, Xiaohui Xie, and Pierre Baldi. Data structures and compression algorithms for high-throughput sequencing technologies. *BMC bioinformatics*, 11(1):514, 2010.

[21] Michael C Davis, Vladímir Bahyl, Germán Cancio, Eric Cano, Julien Leduc, and Steven Murray. Cern tape archive—from development to production deployment. In *EPJ Web of Conferences*, volume 214, page 04015. EDP Sciences, 2019.

[22] Sebastian Deorowicz and Szymon Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8(1):25, 2013.

[23] P. Deutsch and J.-L. Gailly. Rfc1950: Zlib compressed data format specification version 3.3. 1996.

[24] Xavier Espinal, G Adde, B Chan, J Iven, G Lo Presti, M Lamanna, L Mascetti, A Pace, A Peters, S Ponce, et al. Disk storage at cern: Handling lhc data and beyond. In *Journal of Physics: Conference Series*, volume 513, page 042017. IOP Publishing, 2014.

[25] Yuanwei Fang, Chen Zou, and Andrew A Chien. Accelerating raw data analysis with the accorda software and hardware architecture. *Proceedings of the VLDB Endowment*, 12(11):1568–1582, 2019.

[26] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. Udp: a programmable accelerator for extract-transform-load workloads and more. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–68. IEEE, 2017.

[27] Robert M Fano. *The transmission of information.* Massachusetts Institute of Technology, Research Laboratory of Electronics . . . , 1949.

[28] Google. Google cloud pricing, 2020. https://cloud.google.com/pricing.

[29] Google. google/snappy, Apr 2020. `https://github.com/google/snappy`.

[30] L Heinrich and Eric Wulff. Deep autoencoders for data compression in high energy physics. 2019.

[31] Xiaomeng Huang, Yufang Ni, Dexun Chen, Songbin Liu, Haohuan Fu, and Guangwen Yang. Czip: A fast lossless compression algorithm for climate data. *International Journal of Parallel Programming*, 44(6):1248–1267, 2016.

[32] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[33] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[34] Achim; Bin Anuar Afiq Aizuddin; Jomhari, Nur Zulaiha; Geiser. Higgs-to-four-lepton analysis example using 2011-2012 data., 2017. https://cloud.google.com/pricing.

[35] Vardan Khachatryan et al. The CMS trigger system. *JINST*, 12(01):P01020, 2017.

[36] P La Rocca and F Riggi. The upgrade programme of the major experiments at the large hadron collider. In *Journal of Physics: Conference Series*, volume 515, page 012012. IOP Publishing, 2014.

[37] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[38] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[39] Mervat MA Mahmoud, Dalia A El-Dib, and Hossam AH Fahmy. Low energy asic design for main memory data compression/decompression. In *2018 30th International Conference on Microelectronics (ICM)*, pages 256–259. IEEE, 2018.

[40] Jozsef Makai, Andreas Joachim Peters, Georgios Bitzes, Elvin Alin Sindrilaru, Michal Kamil Simon, and Andrea Manzi. Testing of complex, large-scale distributed storage systems: a cern disk storage case study. In *EPJ Web of Conferences*, volume 214, page 05008. EDP Sciences, 2019.

[41] Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. A case for toggle-aware compression for gpu systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–200. IEEE, 2016.

[42] Jim Pivarski, Peter Elmer, and David Lange. Awkward arrays in python, c++, and numba. *arXiv preprint arXiv:2001.06307*, 2020.

[43] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[44] P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142, 2006.

[45] Arjun Rawal, Yuanwei Fang, and Andrew Chien. Programmable acceleration for sparse matrices in a data-movement limited world. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 47–56. IEEE, 2019.

[46] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3454–3463, 2019.

[47] Hervé Rousseau, Belinda Chan Kwok Cheong, Cristian Contescu, Xavier Espinal Curull, Jan Iven, Hugo Gonzalez Labrador, Massimo Lamanna, Giuseppe Lo Presti, Luca

Mascetti, Jakub Moscicki, et al. Providing large-scale disk storage at cern. In *EPJ Web of Conferences*, volume 214, page 04033. EDP Sciences, 2019.

[48] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, page 34–40, New York, NY, USA, 2010. Association for Computing Machinery.

[49] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 67–80, 2016.

[50] Oksana Shadura and Brian Paul Bockelman. Root i/o compression algorithms and their performance impact within run 3. *arXiv preprint arXiv:1906.04624*, 2019.

[51] Claude E Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[52] Gregory Szorc. Better compression with zstandard, Mar 2017. https://gregoryszorc.com/blog/2017/03/07/better-compression-with-zstandard/.

[53] Po-An Tsai and Daniel Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 229–242, New York, NY, USA, 2019. Association for Computing Machinery.

[54] CERN TWiki. 2.2 CMS Computing Model, Jul 2018. `https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookComputingModel`.

[55] Lulu Wen, Kaile Zhou, Shanlin Yang, and Lanlan Li. Compression of smart meter big data: A survey. *Renewable and Sustainable Energy Reviews*, 91:59–69, 2018.

[56] Frans MJ Willems, Yuri M Shtarkov, and Tjalling J Tjalkens. The context-tree weighting method: basic properties. *IEEE transactions on information theory*, 41(3):653–664, 1995.

[57] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[58] Z Zhang and B Bockelman. Exploring compression techniques for root io. In *J. Phys. Conf. Ser.*, volume 898, page 072043, 2017.

[59] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[60] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.