# Exploiting Domain-Specific Data Properties to Improve Compression for High Energy Physics Data

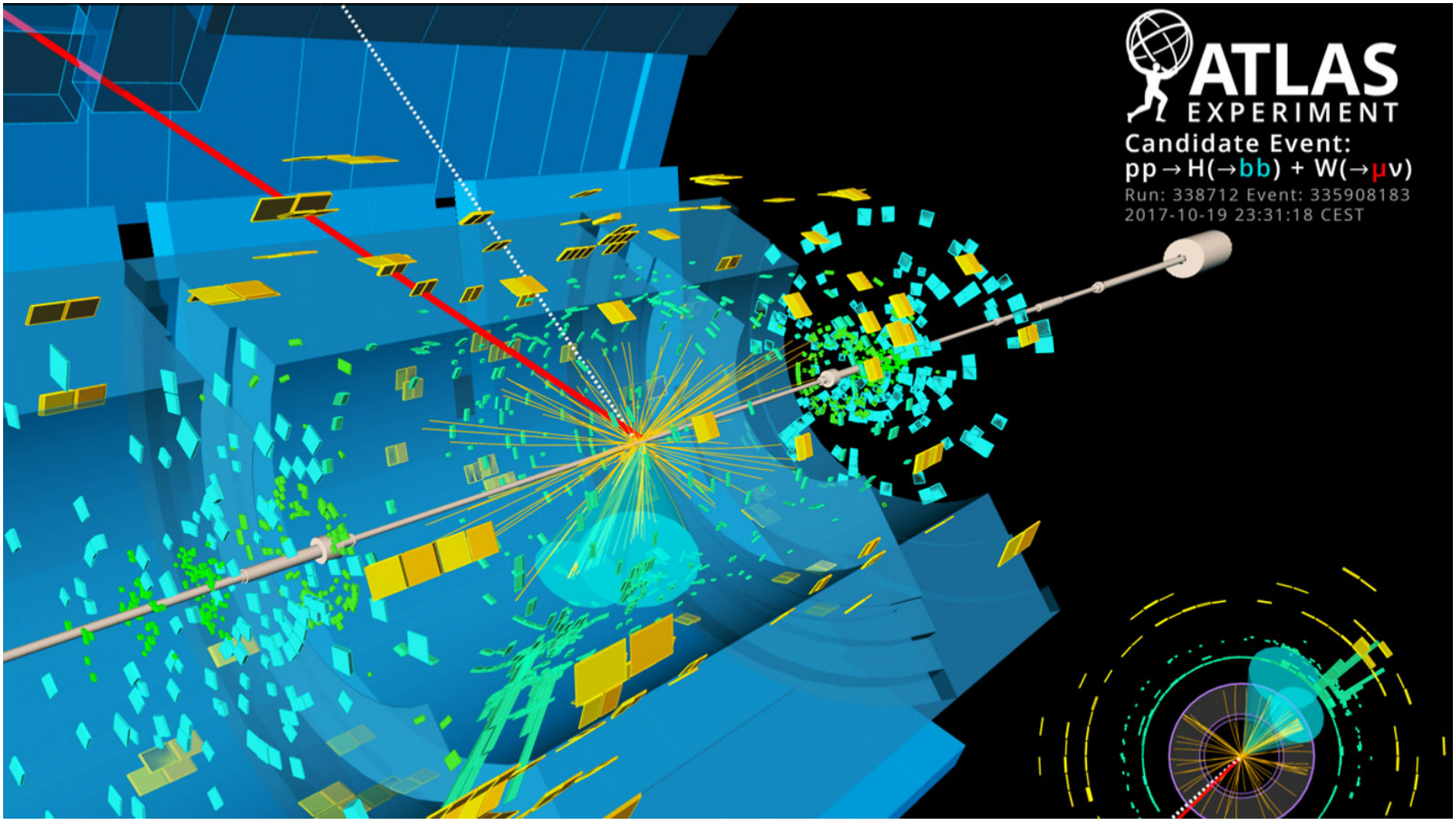
Arjun Rawal

University of Chicago

June 2, 2020

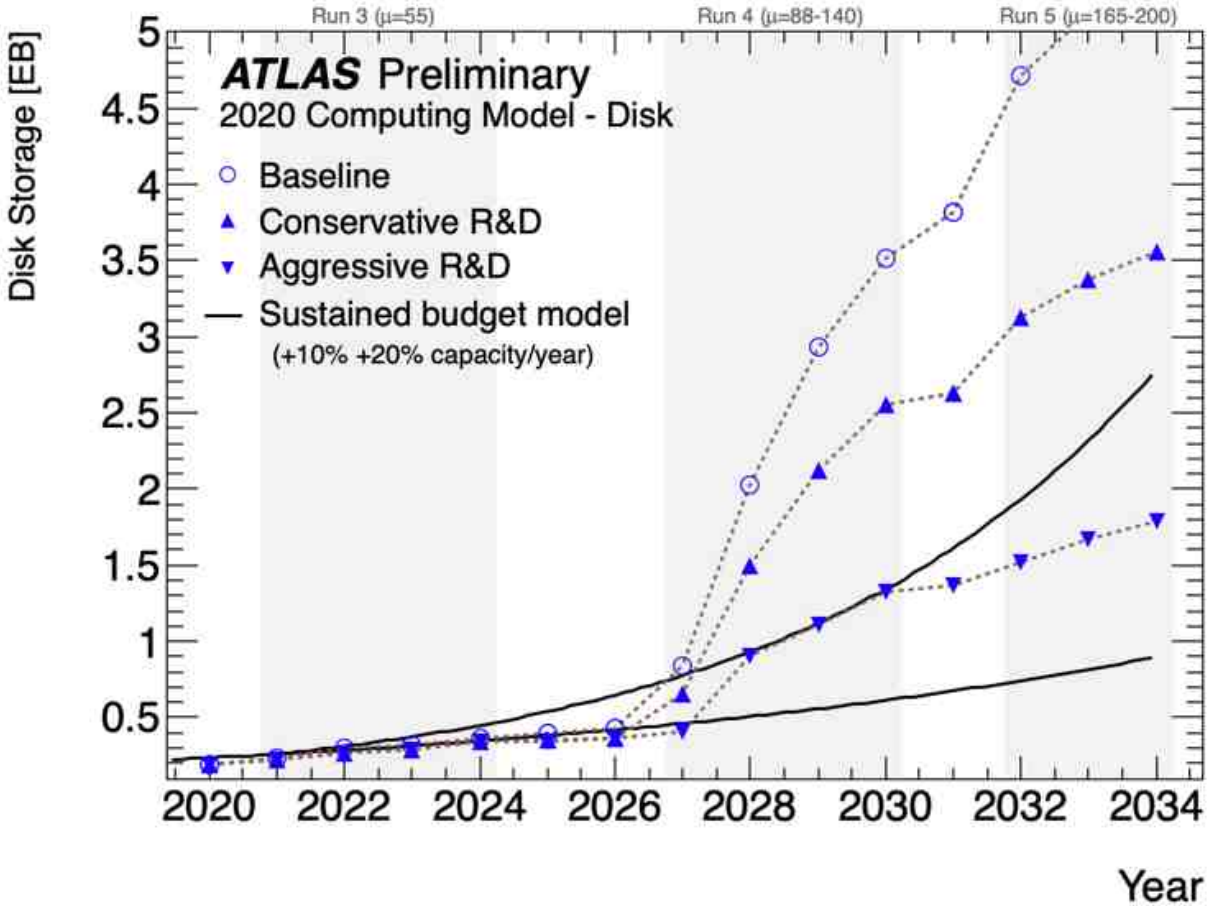
# Context

# High Energy Physics in a Nutshell




Less than $0.1\%$ of data recorded, but still more than $100$ PB per year!

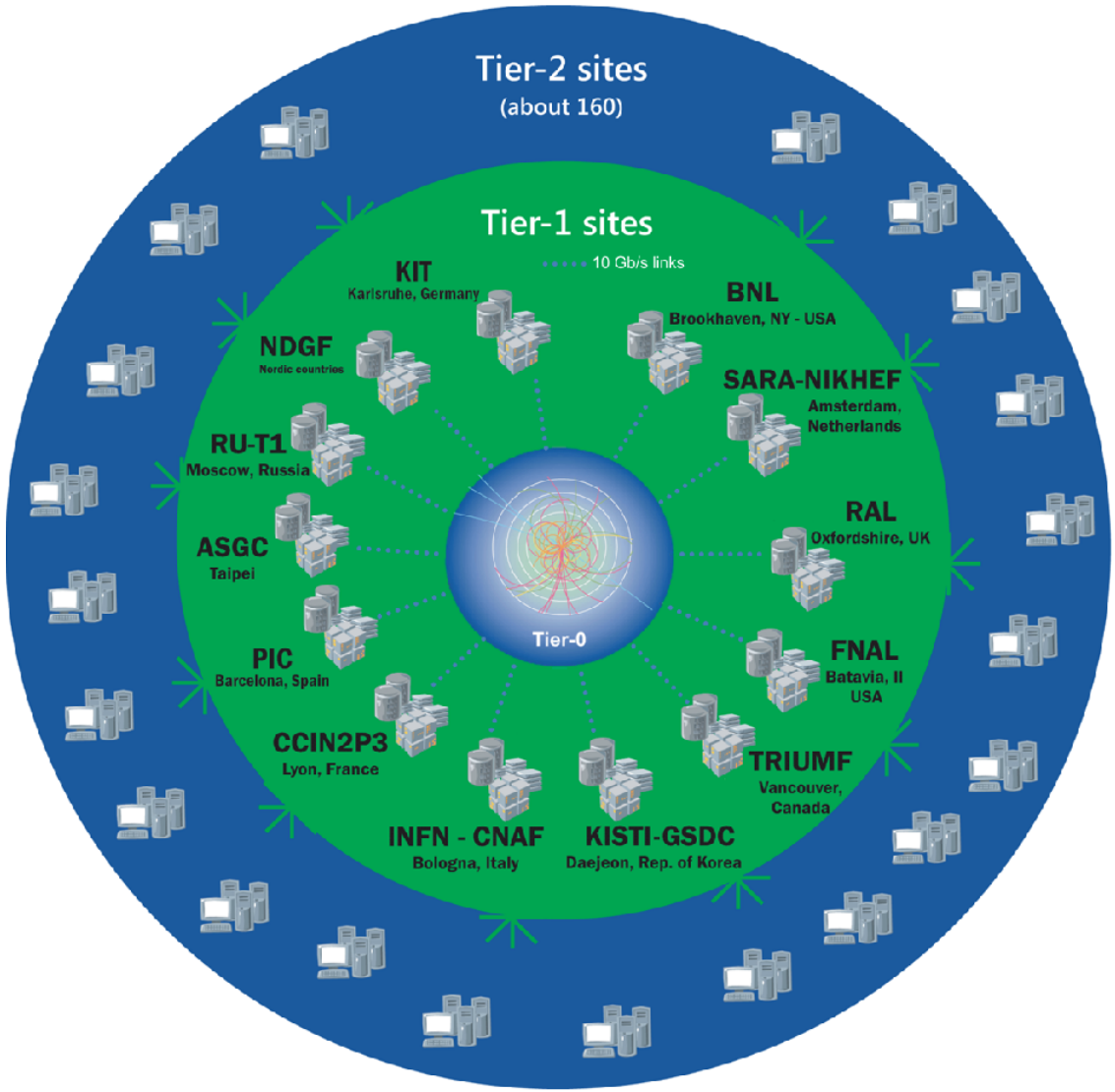
# Projected Data Growth




- Data storage requirements quickly outpace budget constraints.

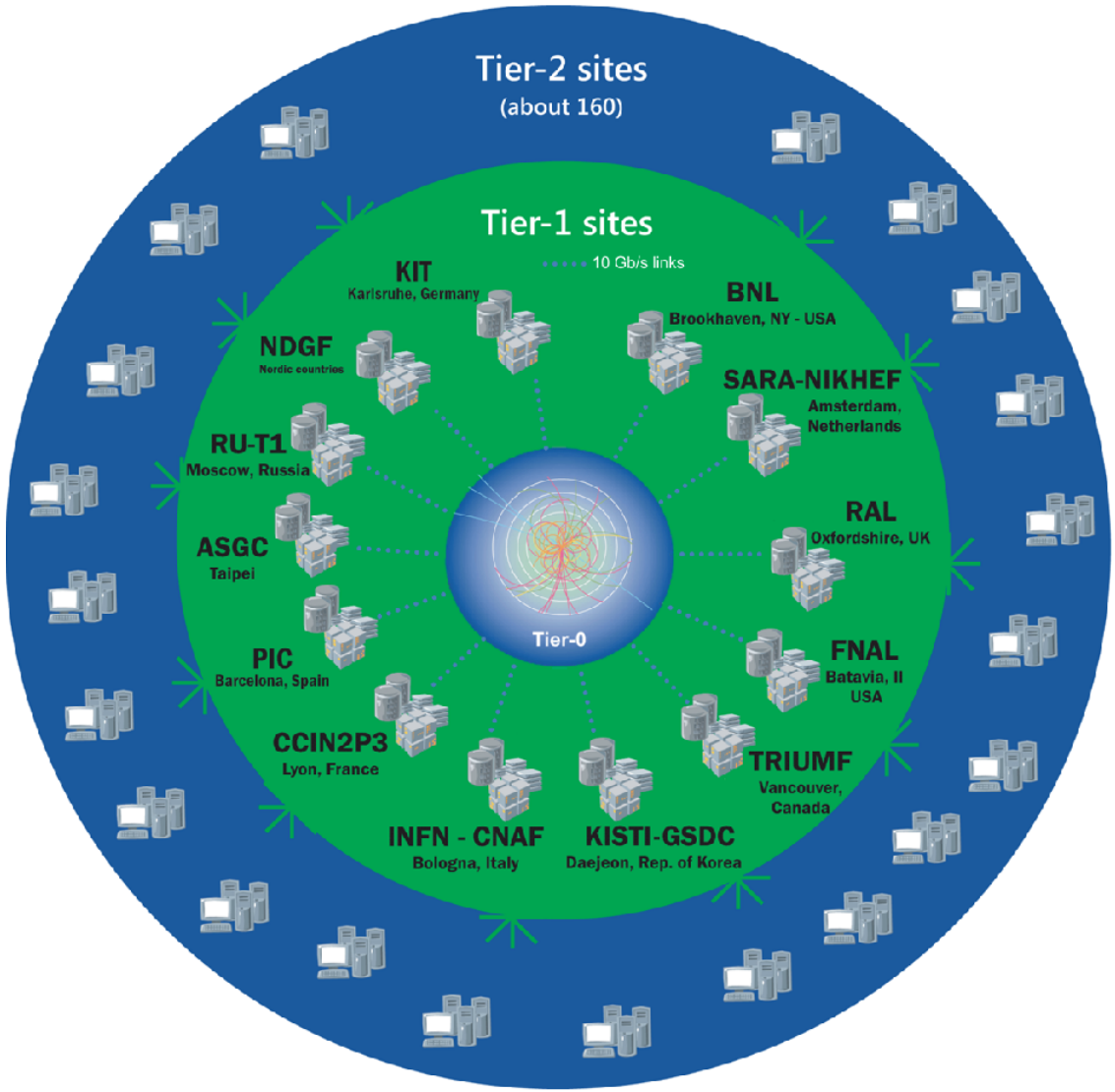
# How Does CERN Store Data?

- Replicated for reliability and access speed
- Data moves from Tier-1 to Tier-2 for analysis and simulation
- Stored using



# How Does CERN Store Data?

- Replicated for reliability and access speed
- Data moves from Tier-1 to Tier-2 for analysis and simulation
- Stored using
  - Tape (350PB)
  - HDD (280PB)
  - SSD (40PB)



# Problem Statement

- High energy physics data is costly to store (exabytes of data, millions of $)
- Current formats prioritize analysis performance over storage size

# Our Approach

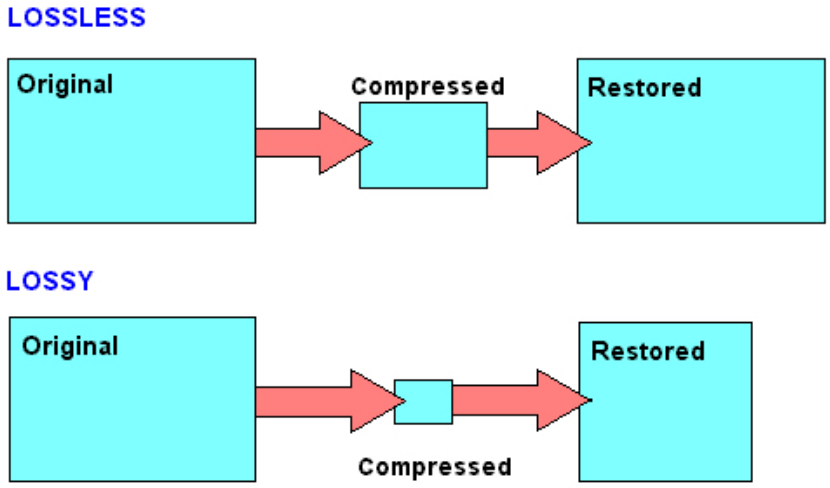
We utilize modern compression algorithms, domain-specific techniques, and aggregated storage to reduce storage requirements for archived high energy physics data.

Furthermore, we show that compression focused storage in a production environment can save petabytes of disk space while only requiring a minimal increase in computational resources.

# Data Compression

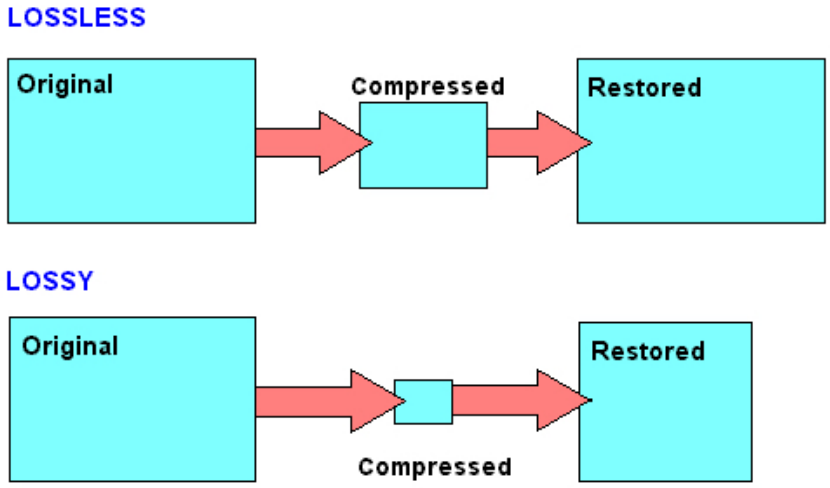
# Data Compression Overview

- **Lossless** vs. Lossy

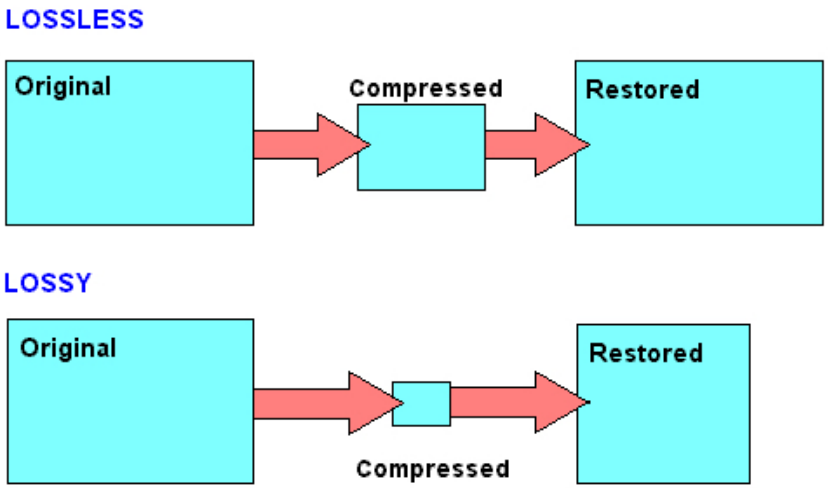


# Data Compression Overview

- **Lossless** vs. Lossy
- Tradeoff between speed & reduction in size



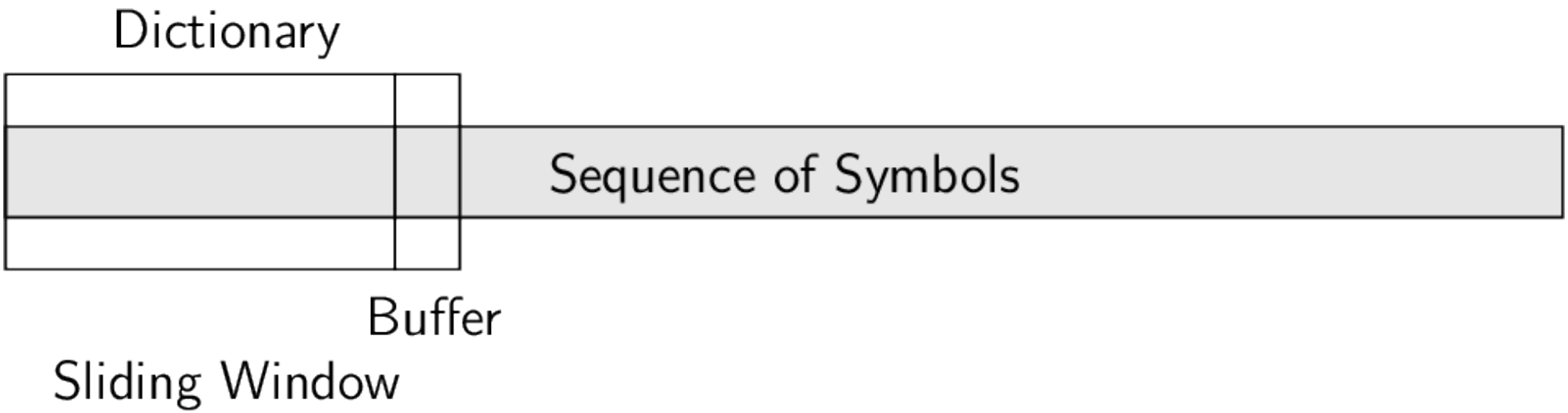
# Data Compression Overview




- **Lossless** vs. Lossy
- Tradeoff between speed & reduction in size
- Common general purpose software
  - **snappy**
  - **zlib**
  - **zstd**
  - lz4
  - lzma
  - brotli

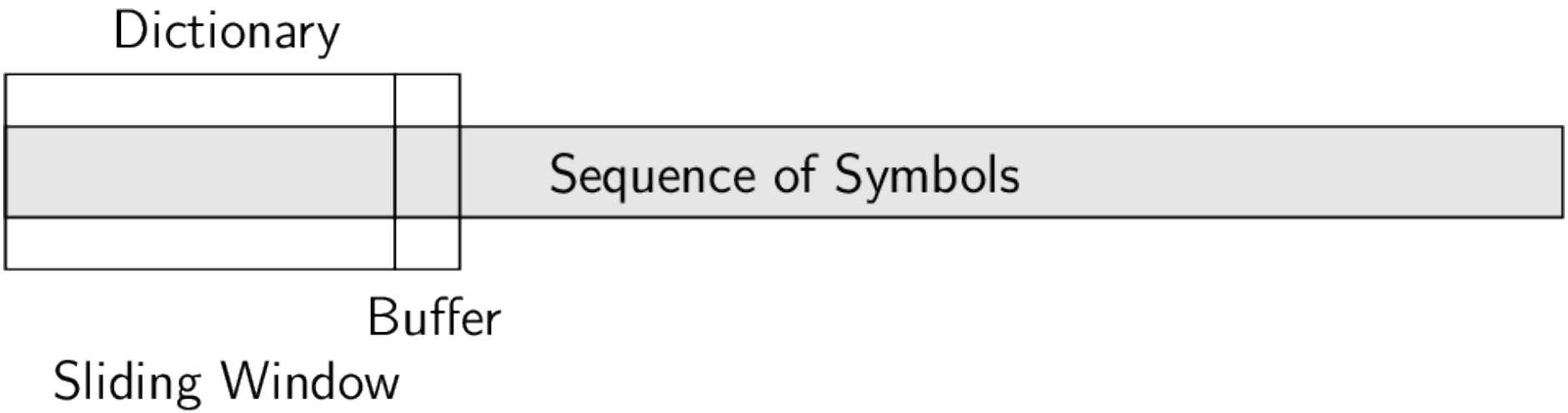
# What Enables Good Compression?

- Repeated Patterns



# What Enables Good Compression?

- Repeated Patterns
- Large Windows



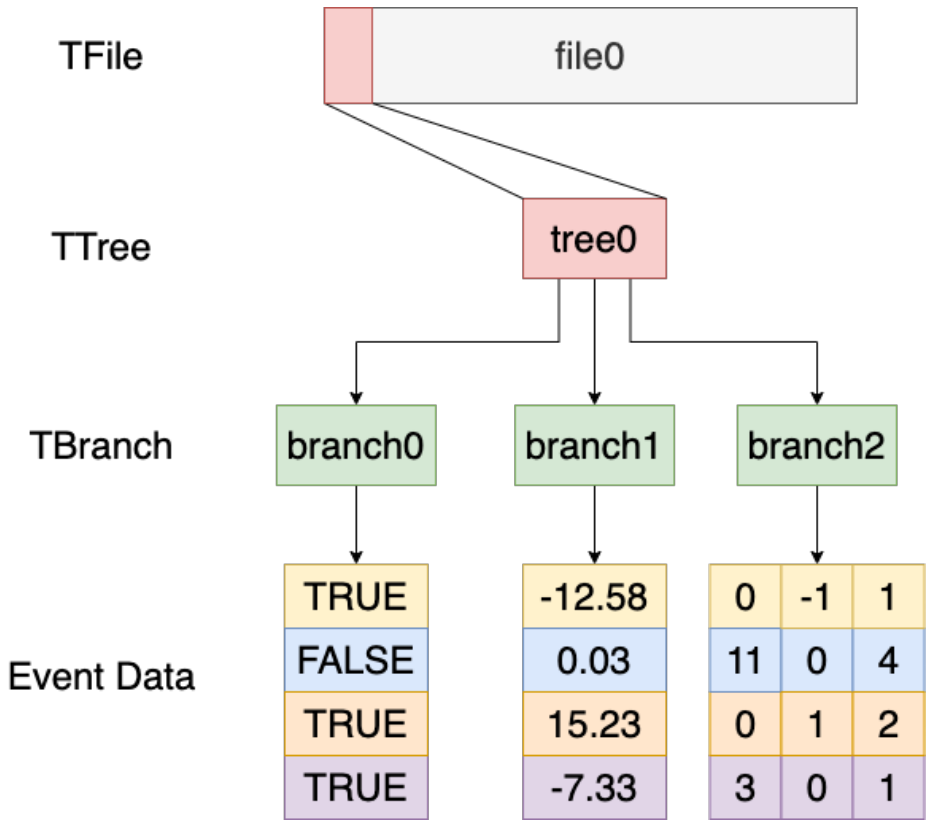
# HEP Data Storage

# The ROOT Framework

- Massive scientific toolkit written in C++
- Used for data processing, statistical analysis, visualisation, and storage

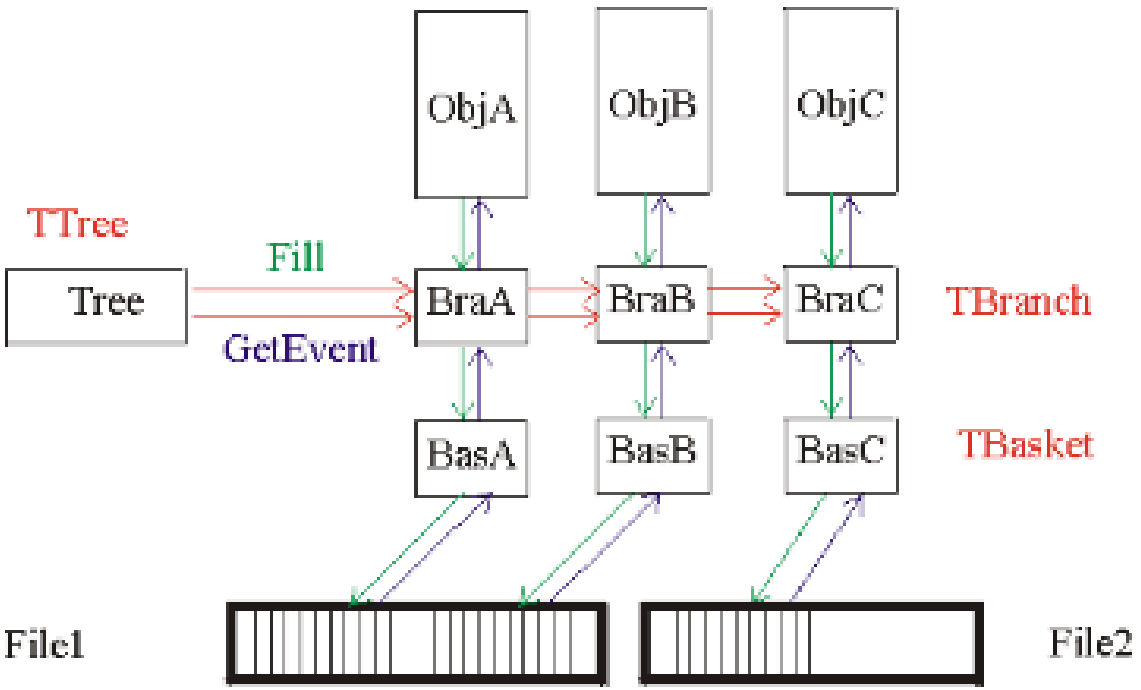
# The ROOT Framework

- Massive scientific toolkit written in C++
- Used for data processing, statistical analysis, visualisation, and storage
- Tree structure for storing data



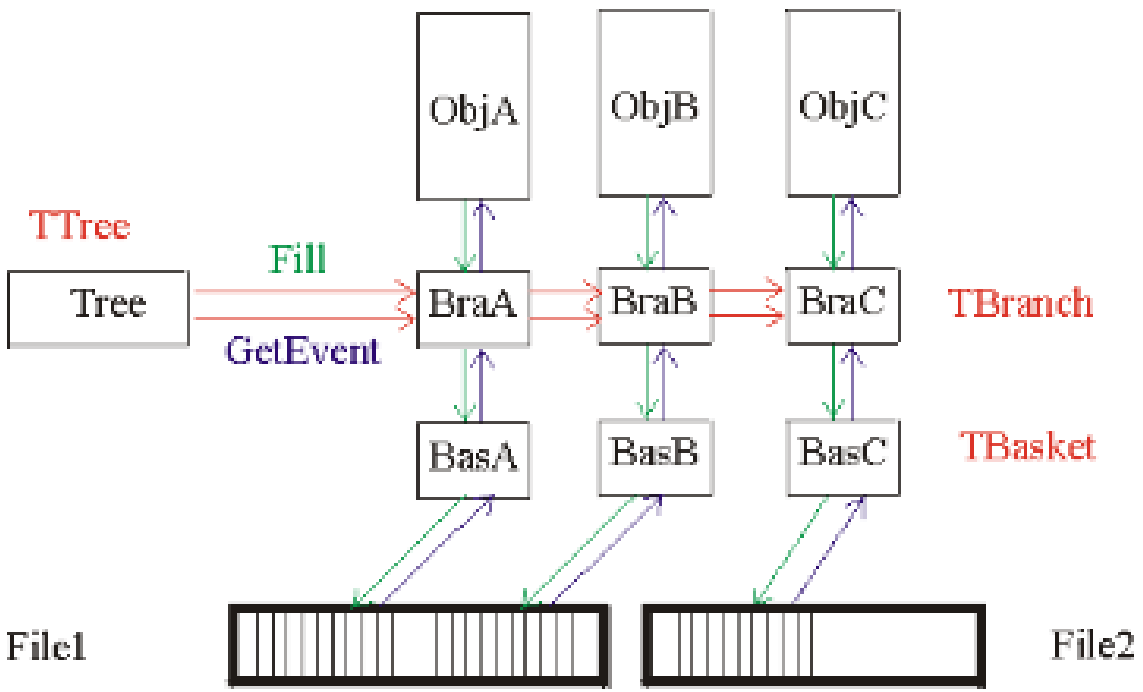
# How Does ROOT Compress Data?

- Basket level compression (`zlib`)
- **Optimized for analysis workloads!**



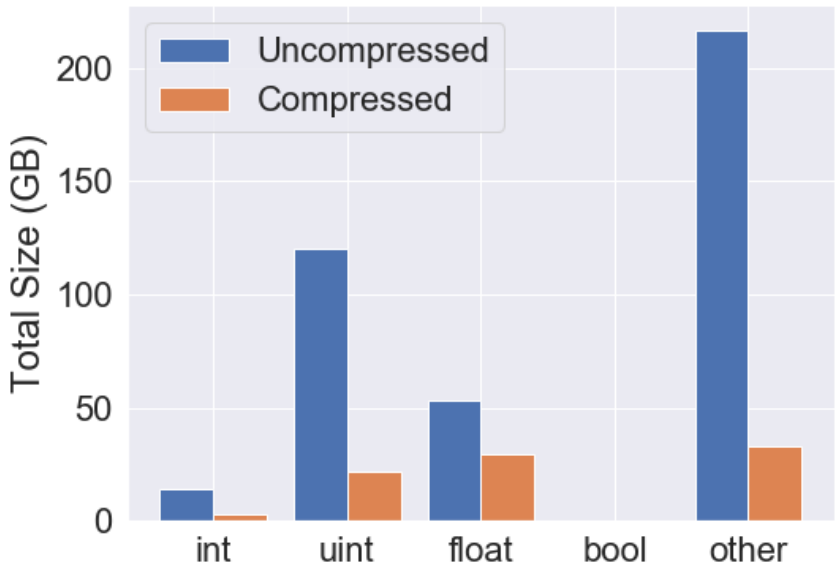
# How Does ROOT Compress Data?

- Basket level compression (`zlib`)
- **Optimized for analysis workloads!**
- Dozens of papers on providing "balance" between size and analysis performance



# What "Kind" of Data is Being Stored?

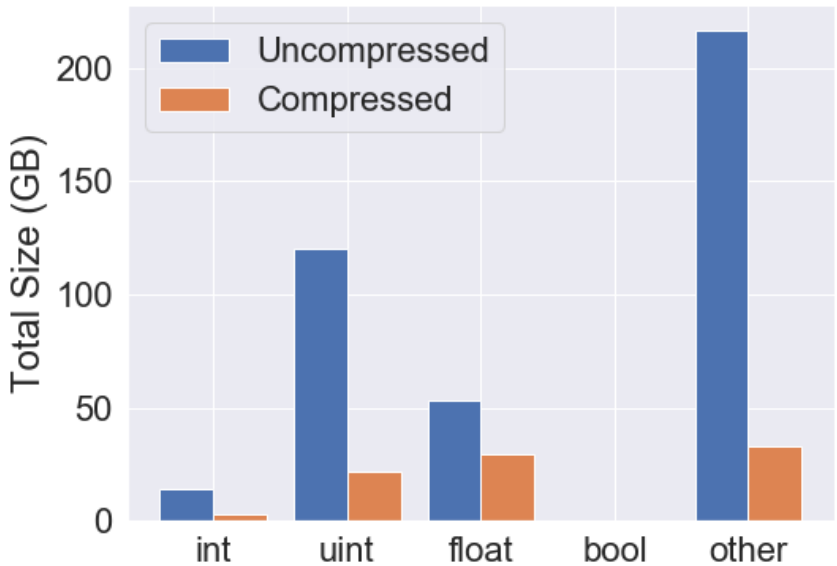
- Current compression ratio $> 5$
- Metadata and structured objects are hard to analyze and easy to compress.




ATLAS Data

# What "Kind" of Data is Being Stored?

- Current compression ratio $> 5$
- Metadata and structured objects are hard to analyze and easy to compress.
- Therefore, we focus on floating point and integer data.




ATLAS Data

# Key Questions

1. What techniques can we use to better compress HEP data?

# Key Questions

1. What techniques can we use to better compress HEP data?
2. How much storage space can we save by applying these techniques?

# Key Questions

1. What techniques can we use to better compress HEP data?
2. How much storage space can we save by applying these techniques?
3. Can this be implemented in a cost effective and scalable way for HEP experiments?

# Approach

# How Do We Reduce Data Storage Needs?

1. Aggregate Data and Use Modern Algorithms

# How Do We Reduce Data Storage Needs?

1. Aggregate Data and Use Modern Algorithms
2. Exploit Patterns and Repetition

# Insight 1: Aggregate Data

- Earlier algorithms like `zlib` have window sizes between 256 bytes and 32 KB

# Insight 1: Aggregate Data

- Earlier algorithms like `zlib` have window sizes between 256 bytes and 32 KB
- Modern algorithms like `zstd` have window sizes between 256 KB and 2GB+

# Insight 1: Aggregate Data

- Earlier algorithms like `zlib` have window sizes between 256 bytes and 32 KB
- Modern algorithms like `zstd` have window sizes between 256 KB and 2GB+
- However, ROOT basket sizes are usually between 8 KB and 8 MB to enable fast data access.

# Insight 1: Aggregate Data

- Earlier algorithms like `zlib` have window sizes between 256 bytes and 32 KB
- Modern algorithms like `zstd` have window sizes between 256 KB and 2GB+
- However, ROOT basket sizes are usually between 8 KB and 8 MB to enable fast data access.

## Key Insight

Larger windows only provide an advantage if data is stored in large enough blocks.

# Insight 2: Exploit Patterns and Repetition

- If we know the type or distribution of a dataset, we can use that information to aid in compression or to pre-train an algorithm.
- These techniques may not provide data reduction on their own.

# Insight 2: Exploit Patterns and Repetition

- If we know the type or distribution of a dataset, we can use that information to aid in compression or to pre-train an algorithm.
- These techniques may not provide data reduction on their own.

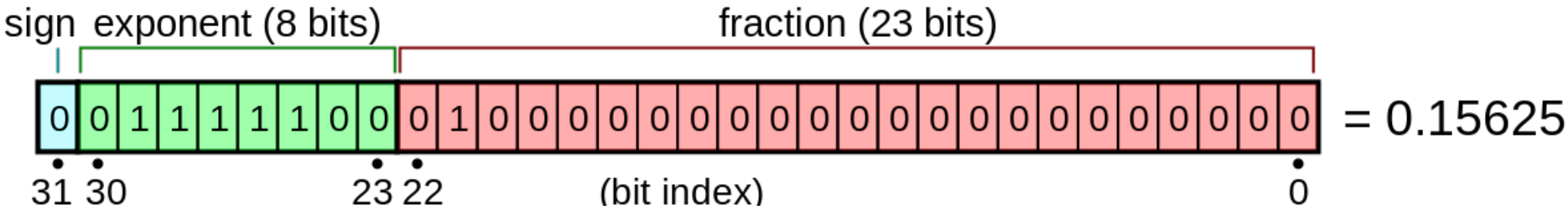
## Delta Encoding

Take the difference between successive elements:

Original Datastream: `[1,2,3,4,5,7,9,8,...]`

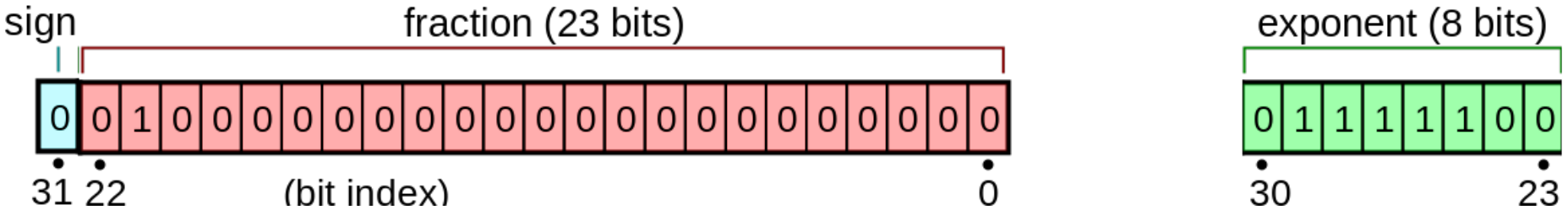
$\delta$-encoded Datastream: `[1,1,1,1,1,2,2,-1,...]`

# Insight 2: Exploit Patterns and Repetition



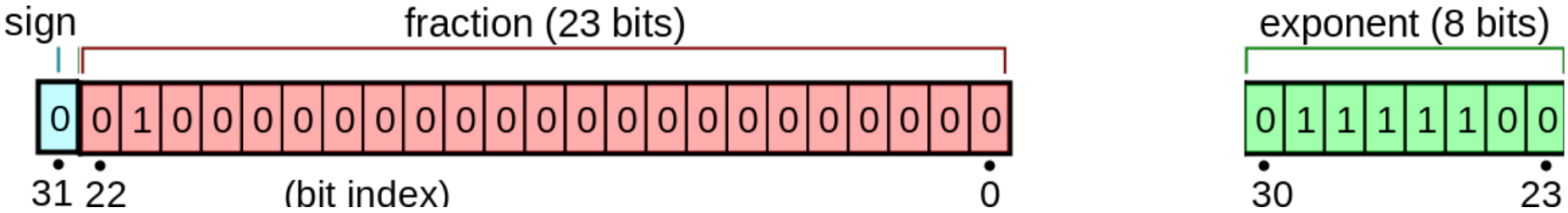

- Sign and exponent portions are more likely to be similar across values.

# Insight 2: Exploit Patterns and Repetition




- Sign and exponent portions are more likely to be similar across values.
- Single precision floating point values can be “split”.

# Insight 2: Exploit Patterns and Repetition




- Sign and exponent portions are more likely to be similar across values.
- Single precision floating point values can be "split".

## Key Insight

By aligning the same components across values, we increase the similarity within blocks.

# Improving CR (Specified)

## Approach

We utilize modern compression algorithms, domain-specific techniques, and aggregated storage to reduce storage requirements for archived high energy physics data.

# Improving CR (Specified)

## Approach

We utilize **modern compression algorithms**, domain-specific techniques, and aggregated storage to reduce storage requirements for archived high energy physics data.

1. Move from `zlib` to `zstd`

# Improving CR (Specified)

## Approach

We utilize modern compression algorithms, **domain-specific techniques**, and aggregated storage to reduce storage requirements for archived high energy physics data.

1. Move from `zlib` to `zstd`
2. Delta encoding and float splitting

# Improving CR (Specified)

## Approach

We utilize modern compression algorithms, domain-specific techniques, and **aggregated storage** to reduce storage requirements for archived high energy physics data.

1. Move from `zlib` to `zstd`
2. Delta encoding and float splitting
3. Compress on branch level granularity

# Improving CR (Specified)

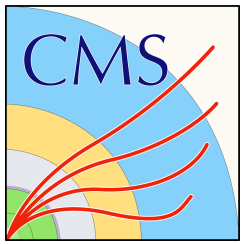
## Approach

We utilize modern compression algorithms, domain-specific techniques, and aggregated storage to reduce storage requirements for archived high energy physics data.

1. Move from `zlib` to `zstd`
2. Delta encoding and float splitting
3. Compress on branch level granularity
4. Pretrain compression dictionaries

# Evaluation

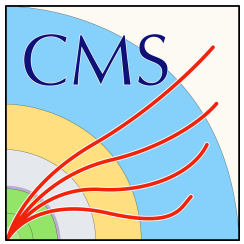
# CMS+ATLAS Experiments/Methodology





- Largest HEP data release in history (1PB)
- Mix of end user data and analysis objects

# CMS+ATLAS Experiments/Methodology



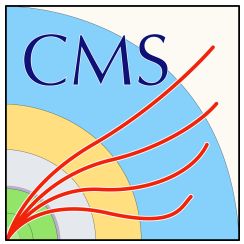


- Largest HEP data release in history (1PB)
- Mix of end user data and analysis objects

---

- Not feasible to experiment on petabytes of data $\rightarrow$ Sampling

# CMS+ATLAS Experiments/Methodology



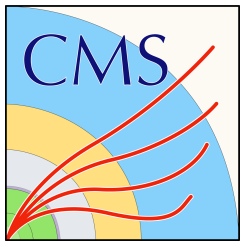


- Largest HEP data release in history (1PB)
- Mix of end user data and analysis objects

---

- Not feasible to experiment on petabytes of data $\rightarrow$ Sampling
- **Extract** data and then evaluate compression

# CMS+ATLAS Experiments/Methodology





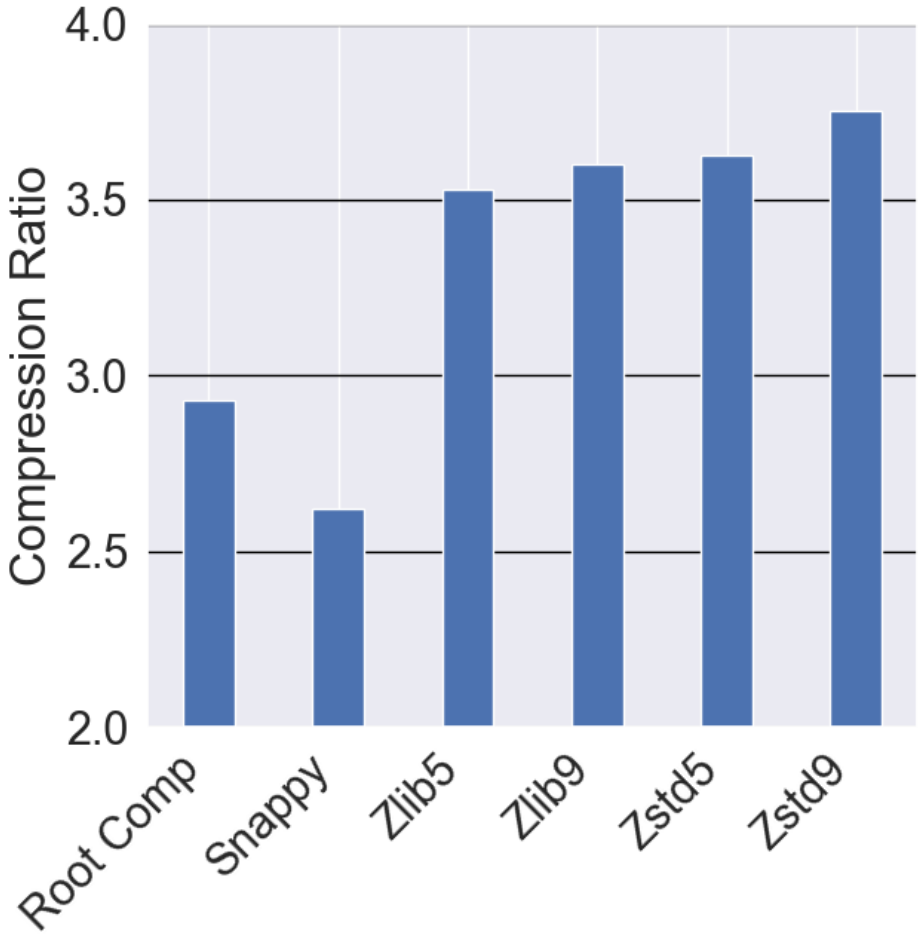
- Largest HEP data release in history (1PB)
- Mix of end user data and analysis objects

---

- Not feasible to experiment on petabytes of data $\rightarrow$ Sampling
- **Extract** data and then evaluate compression
- **Compression Ratio (CR)**: $\frac{\text{Uncompressed Size}}{\text{Compressed Size}}$, reflects how compressed data is from the original.

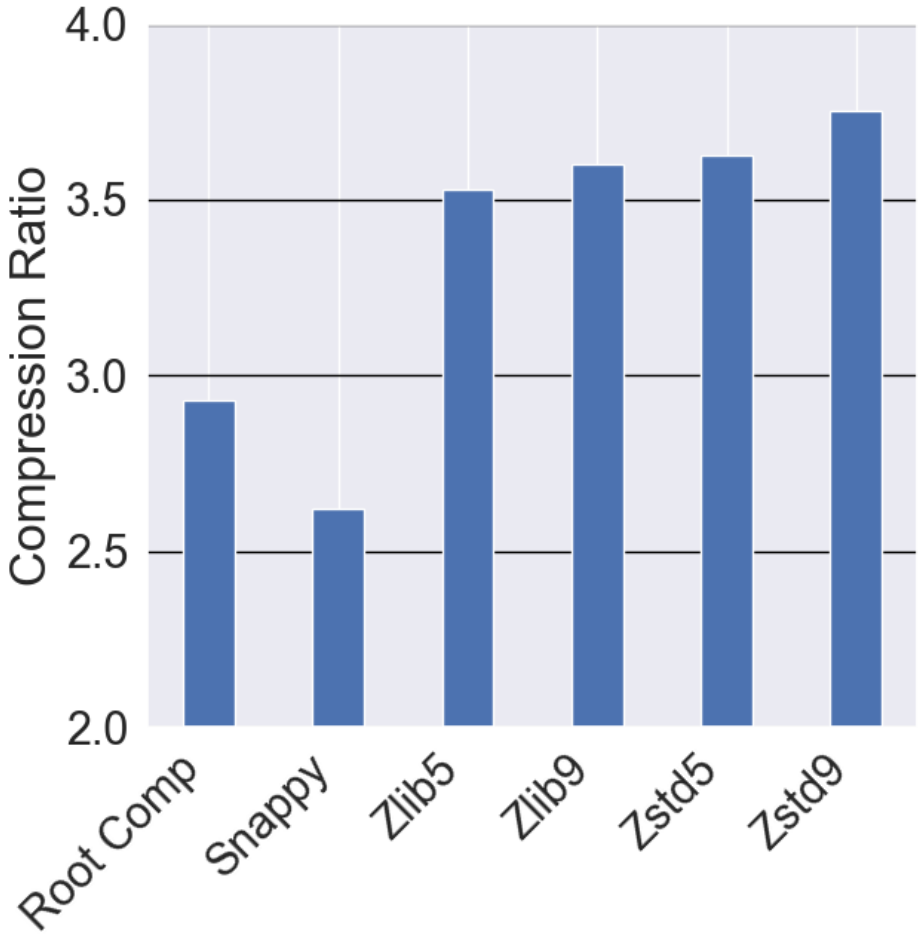
# Better Compression Algorithms

- Compression level is a proxy for resource utilization (higher = more resources)
- `zstd` delivers $> 5\%$ better compression ratio than `zlib`



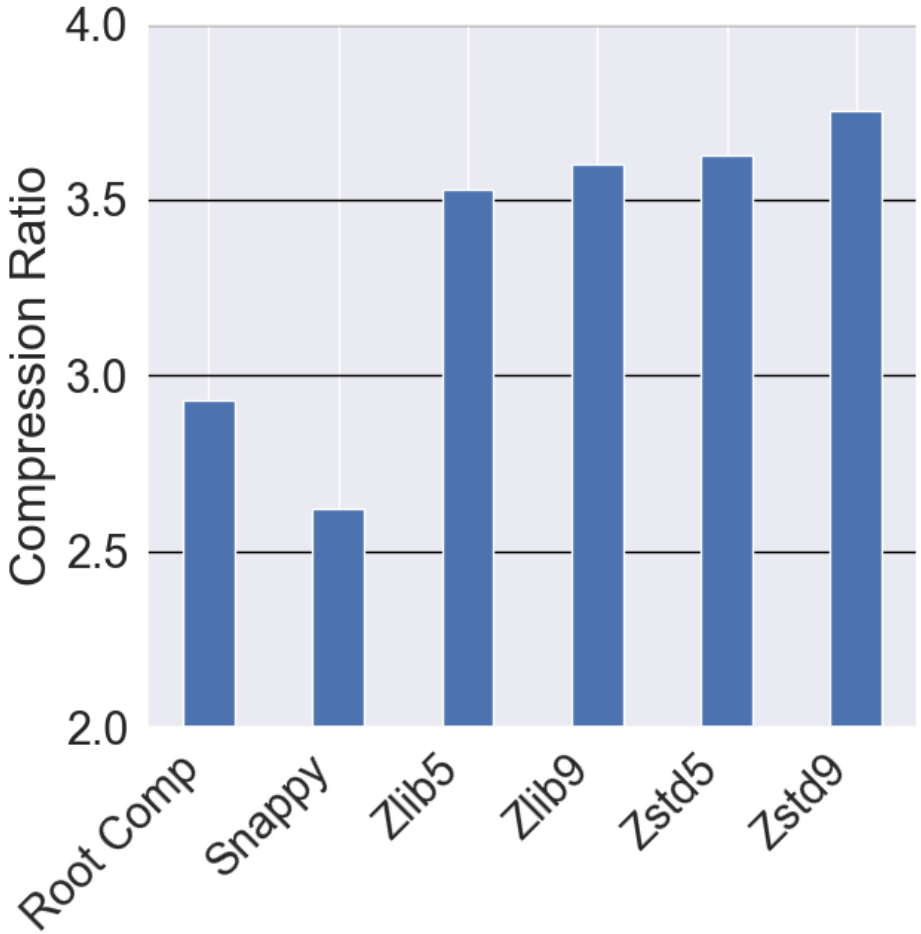
# Better Compression Algorithms

- Compression level is a proxy for resource utilization (higher = more resources)
- `zstd` delivers $> 5\%$ better compression ratio than `zlib`
- `zlib`, level=5 significantly outperforms ROOT compression.



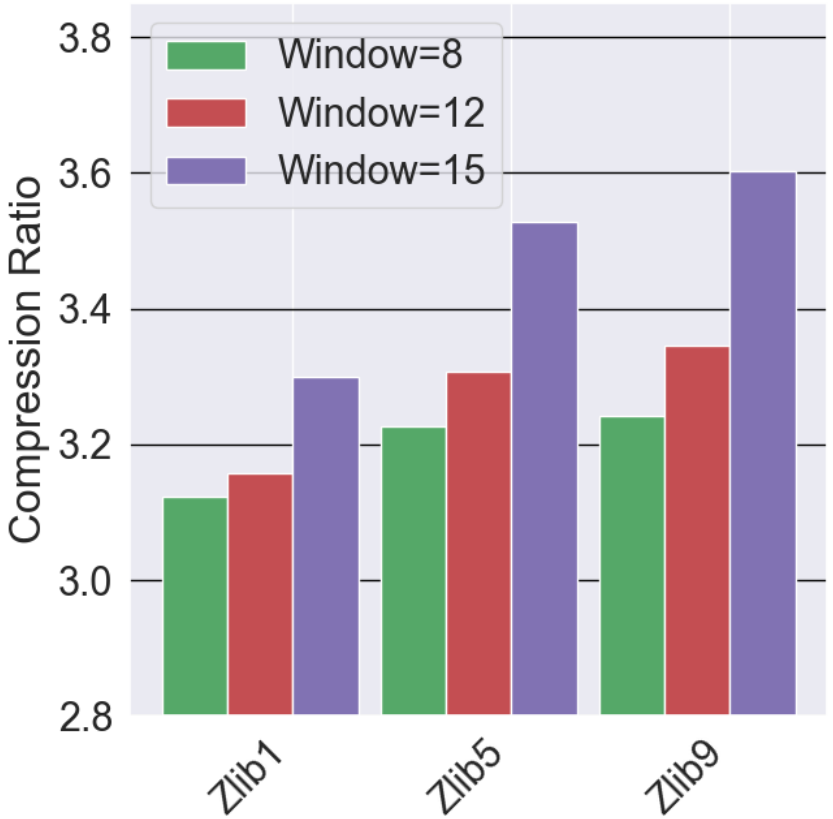
# Better Compression Algorithms

- Compression level is a proxy for resource utilization (higher = more resources)
- `zstd` delivers $> 5\%$ better compression ratio than `zlib`
- `zlib`, level=5 significantly outperforms ROOT compression. **Why?**



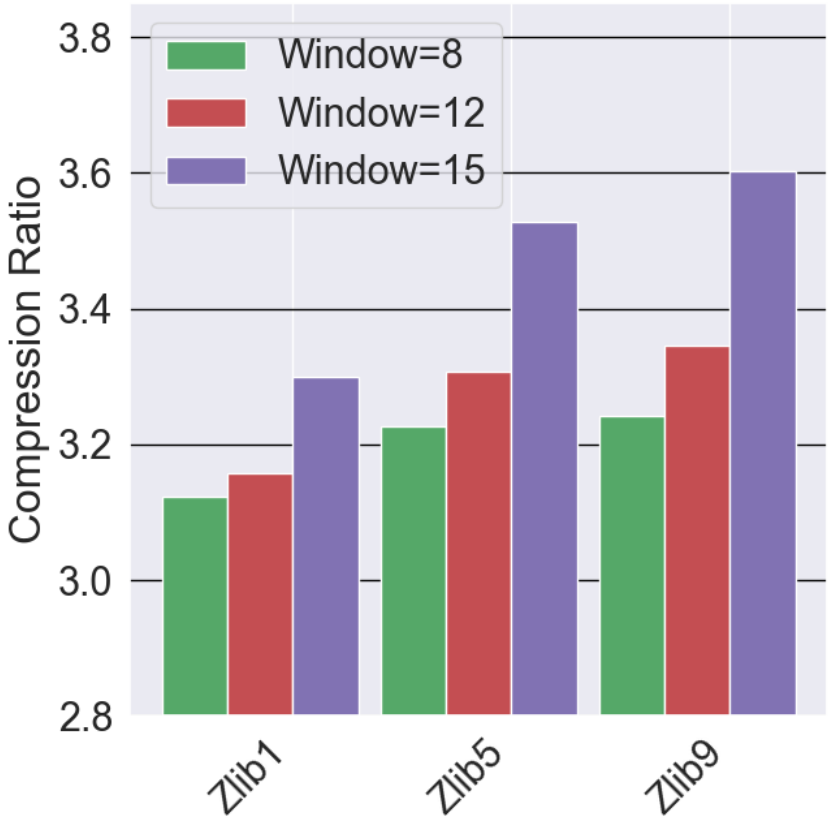
# Exploit Data Layout

- Window size is the amount of data the algorithm can view
- ROOT basket sizes can range from $2^{10}$ to $2^{22}$ bytes, but branches are between $2^{12}$ and $2^{30}$ bytes



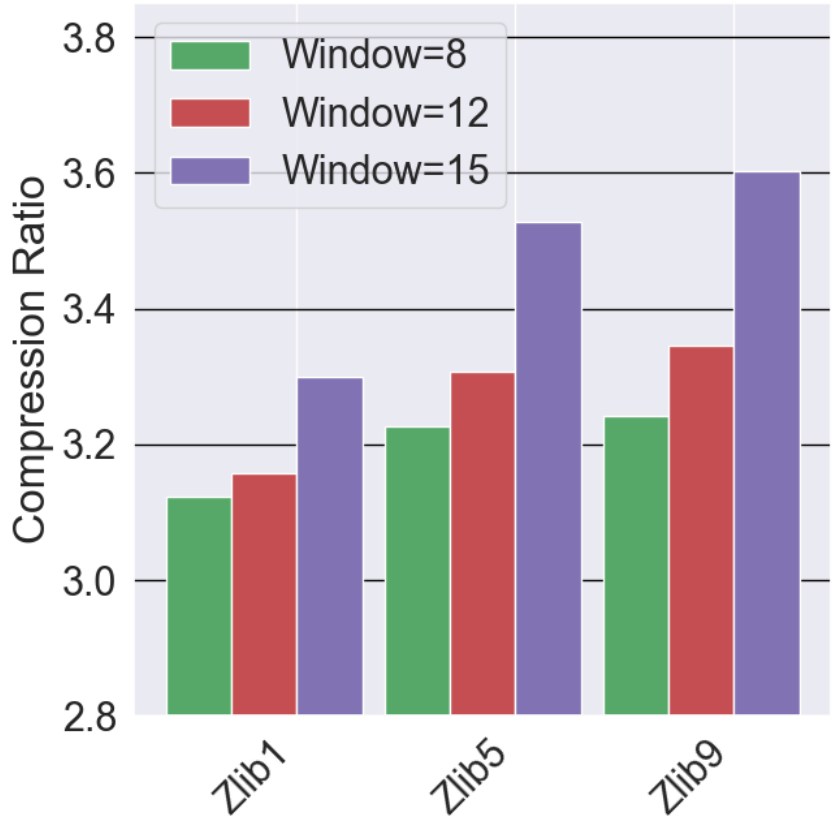
# Exploit Data Layout

- Window size is the amount of data the algorithm can view
- ROOT basket sizes can range from $2^{10}$ to $2^{22}$ bytes, but branches are between $2^{12}$ and $2^{30}$ bytes
- `zstd` windows can be up to $2^{31}$ bytes



# Exploit Data Layout

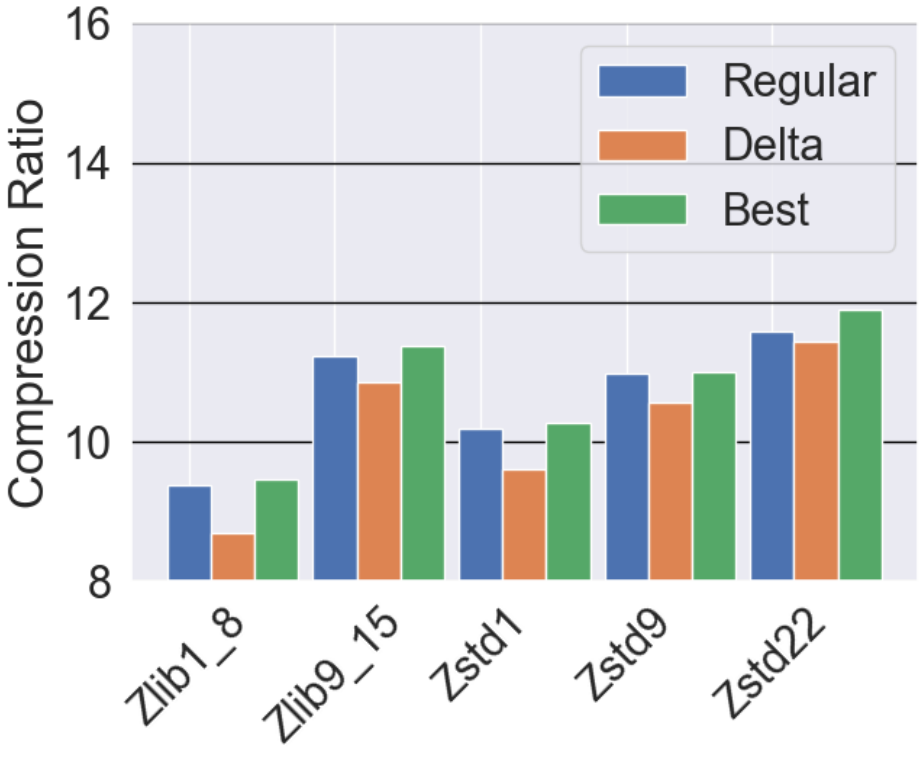
- Window size is the amount of data the algorithm can view
- ROOT basket sizes can range from $2^{10}$ to $2^{22}$ bytes, but branches are between $2^{12}$ and $2^{30}$ bytes
- `zstd` windows can be up to $2^{31}$ bytes



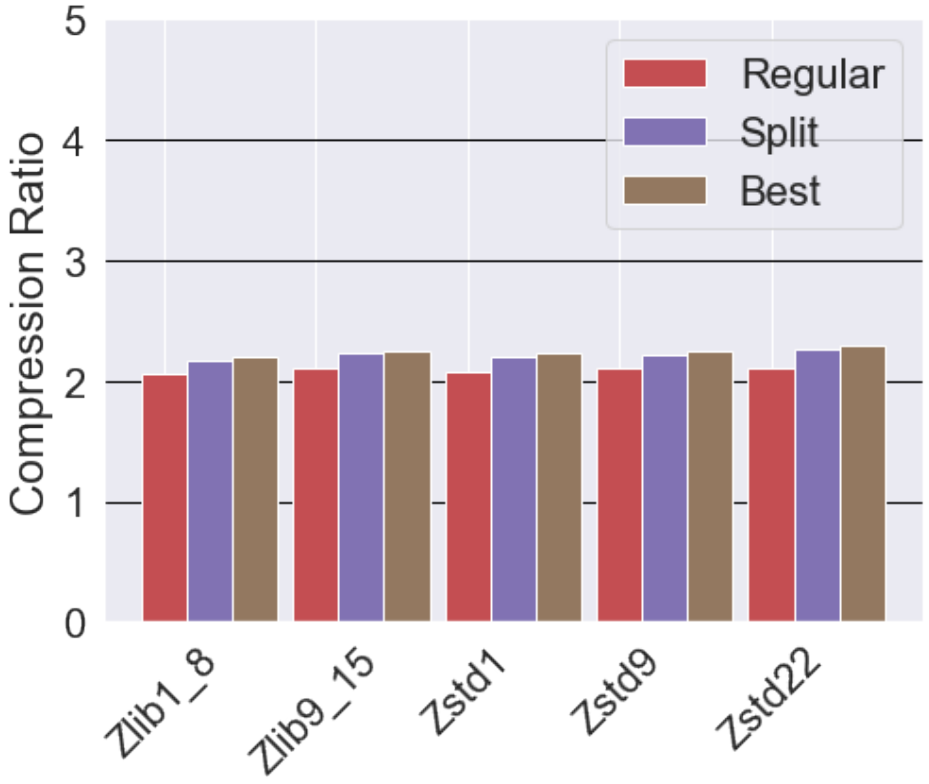

## Conclusion

Compressing at the granularity of branches enables better CR (full windows).

# Delta Encoding and Float Splitting

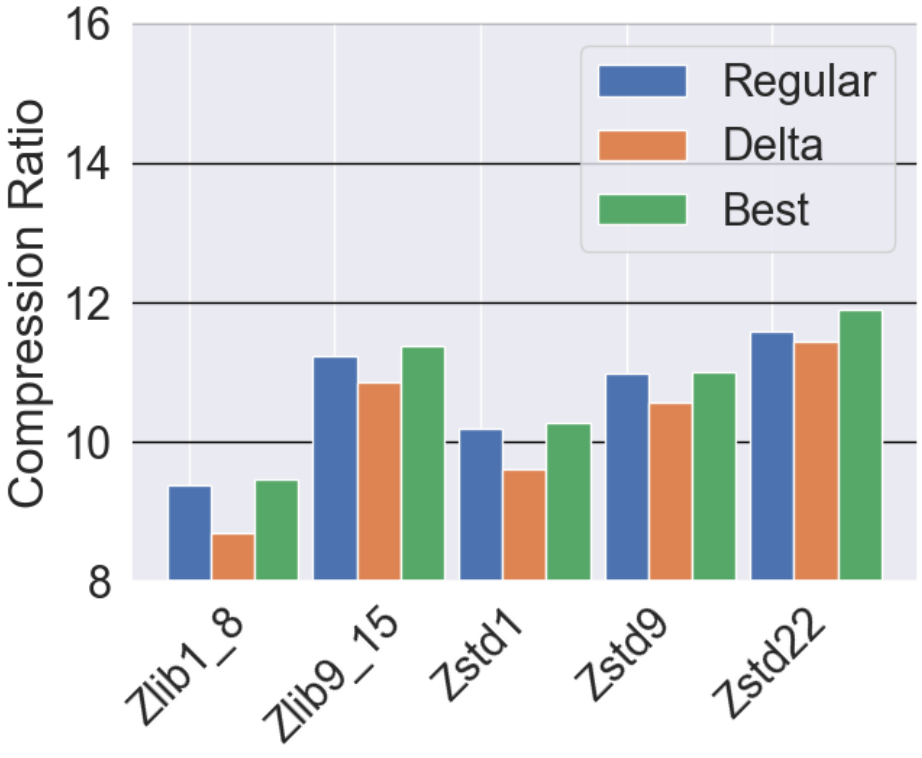



Integer data

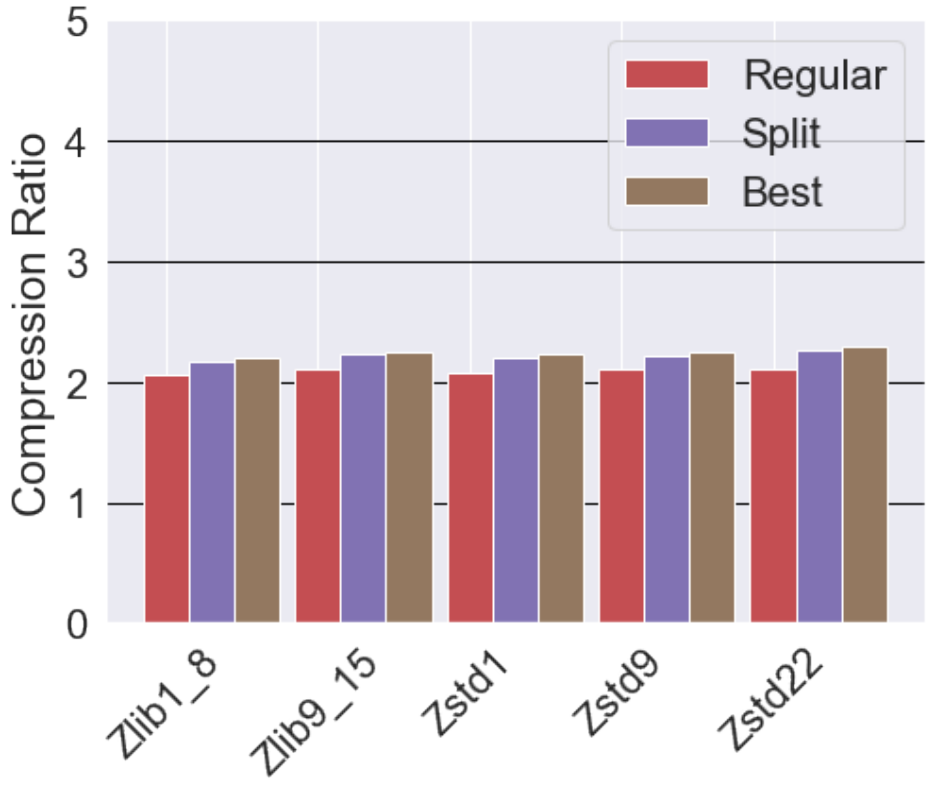



Floating point data

# Delta Encoding and Float Splitting




Integer data

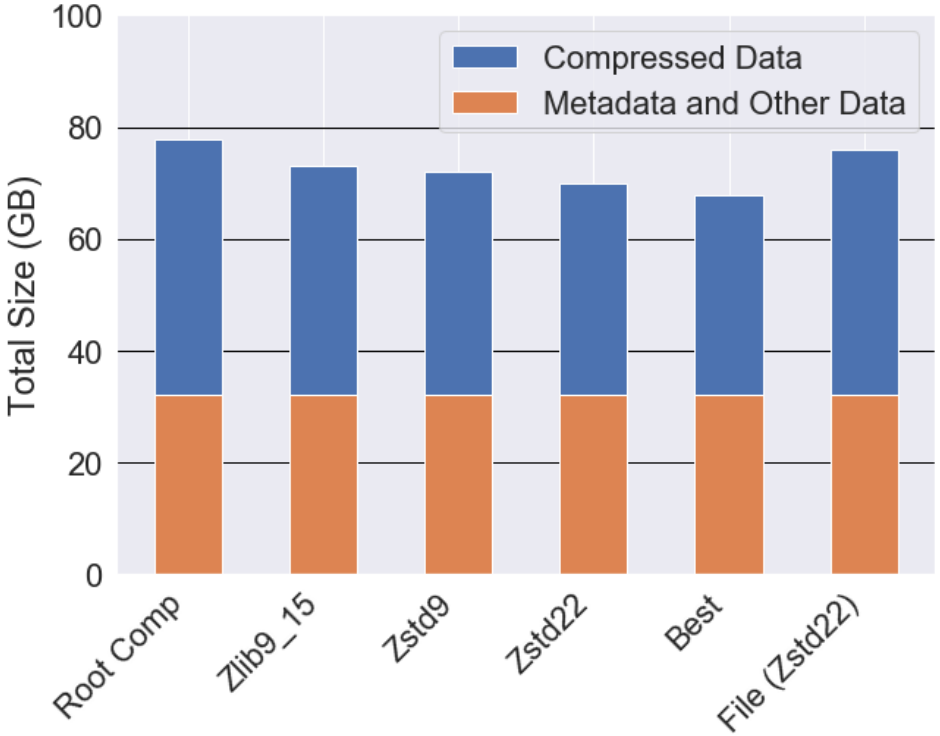



Floating point data

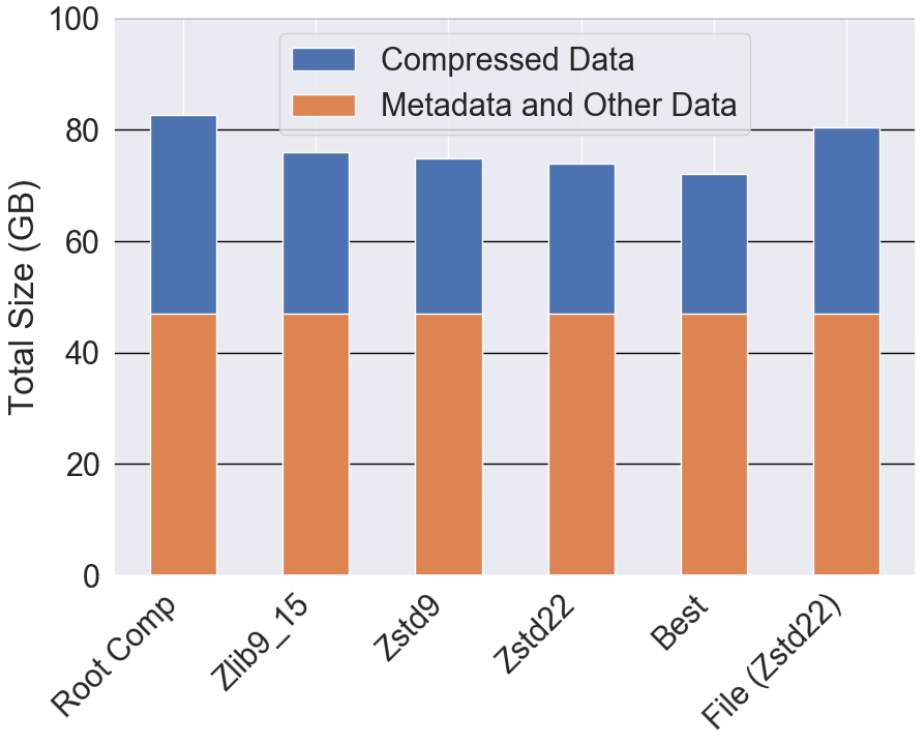
## Conclusion

Delta compression, used selectively, improves integer CR by 5%.
Floating point splitting, used selectively, improves float CR by 4%.

# General Takeaway

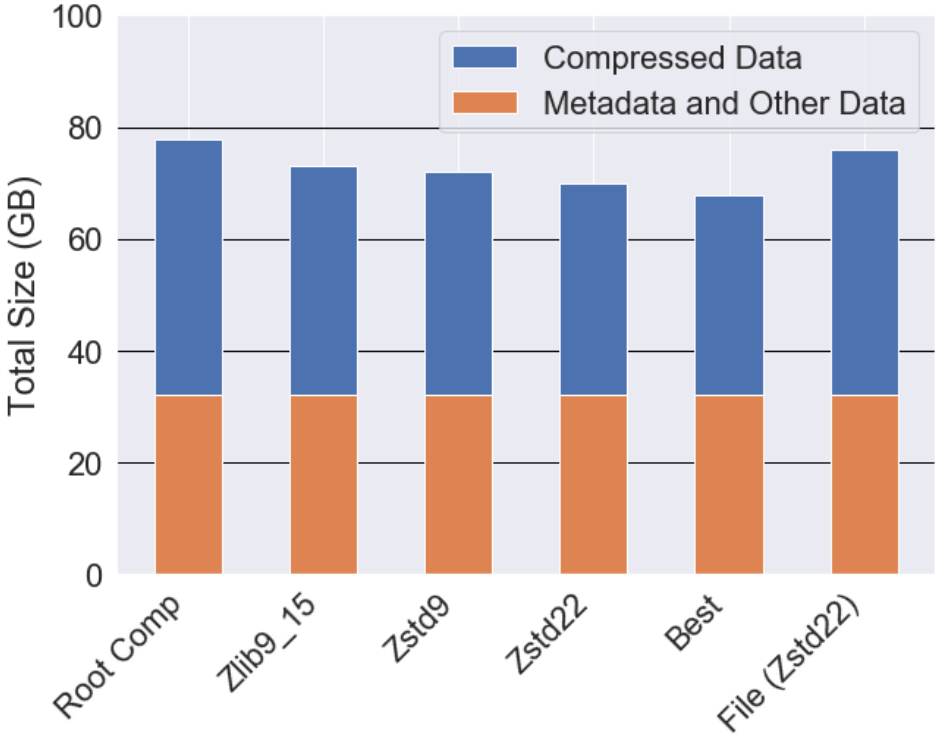



CMS Data

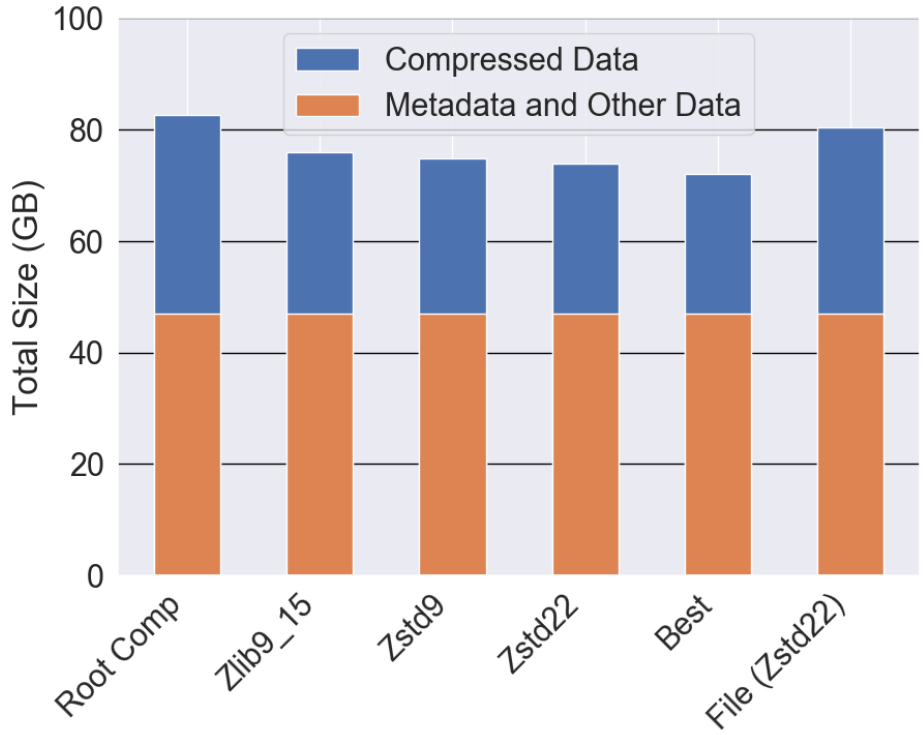



ATLAS Data

# General Takeaway




CMS Data




ATLAS Data

## Conclusion

We reduce data storage usage by up to 10% with `zstd`, level=9 and 15% with `zstd`, level=22.

# Compression at Production Scale

# Insight: Decompression >> Compression

- Common data use case is Filtering $\rightarrow$ Selection $\rightarrow$ Visualization
- HEP data access is mainly $> 95\%$ **reads**.
- Data can be compressed offline, but **must** be decompressed immediately when needed for analysis.

# Insight: Decompression >> Compression

- Common data use case is Filtering $\rightarrow$ Selection $\rightarrow$ Visualization
- HEP data access is mainly $> 95\%$ **reads**.
- Data can be compressed offline, but **must** be decompressed immediately when needed for analysis.

## Key Insight

As long as decompression is done quickly, slower compression is not a barrier to analysis performance.

# Intuition and Context

- Data storage needs expected to grow rapidly

# Intuition and Context

- Data storage needs expected to grow rapidly
- Use better algorithms, compression techniques, and aggregation to get better compression ratio

# Intuition and Context

- Data storage needs expected to grow rapidly
- Use better algorithms, compression techniques, and aggregation to get better compression ratio
- To enable good analysis performance, data reads must be fast

# Intuition and Context

- Data storage needs expected to grow rapidly
- Use better algorithms, compression techniques, and aggregation to get better compression ratio
- To enable good analysis performance, data reads must be fast

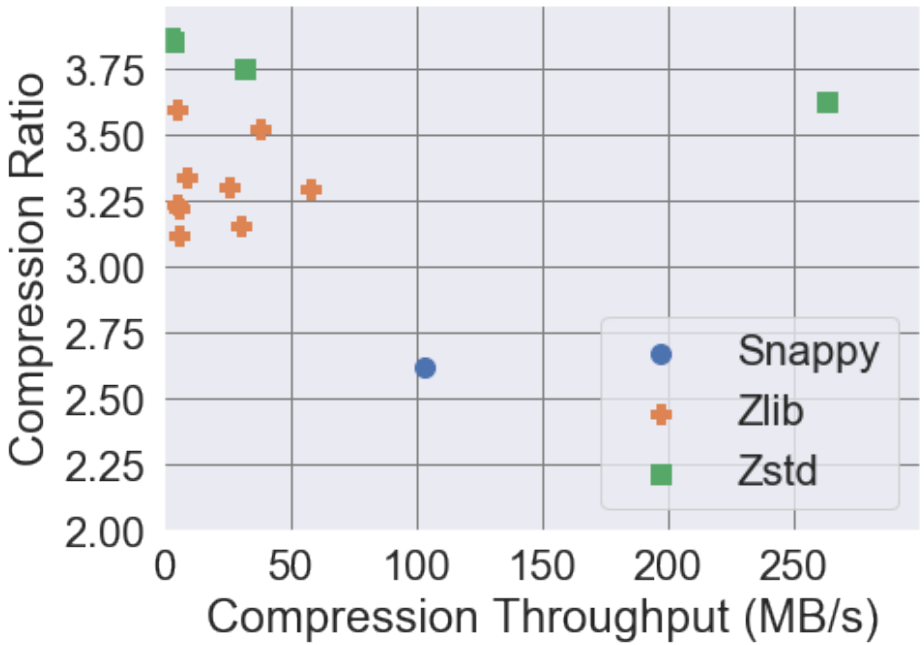
## Scaling Up to Production

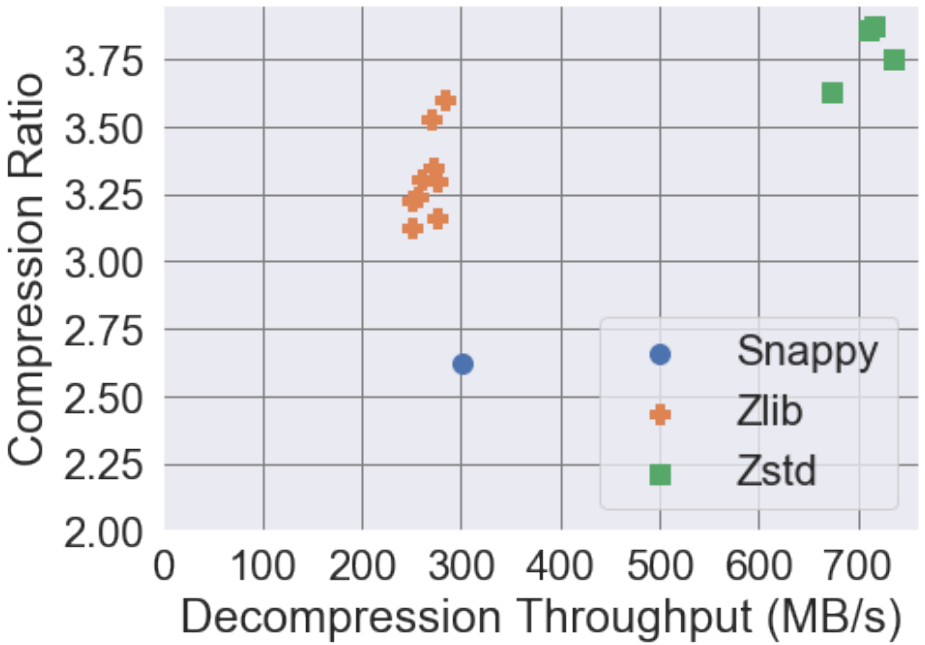
Design compression strategies to reduce data storage requirements, while delivering good read performance.

# Metrics

- **Compression Throughput**: $\frac{\text{Uncompressed Size}}{\text{Time to Compress}}$, shows single core throughput.
- **Core-Hour**: The amount of work done in one hour by a single core on the evaluation system (2.8 GHz Intel Xeon).

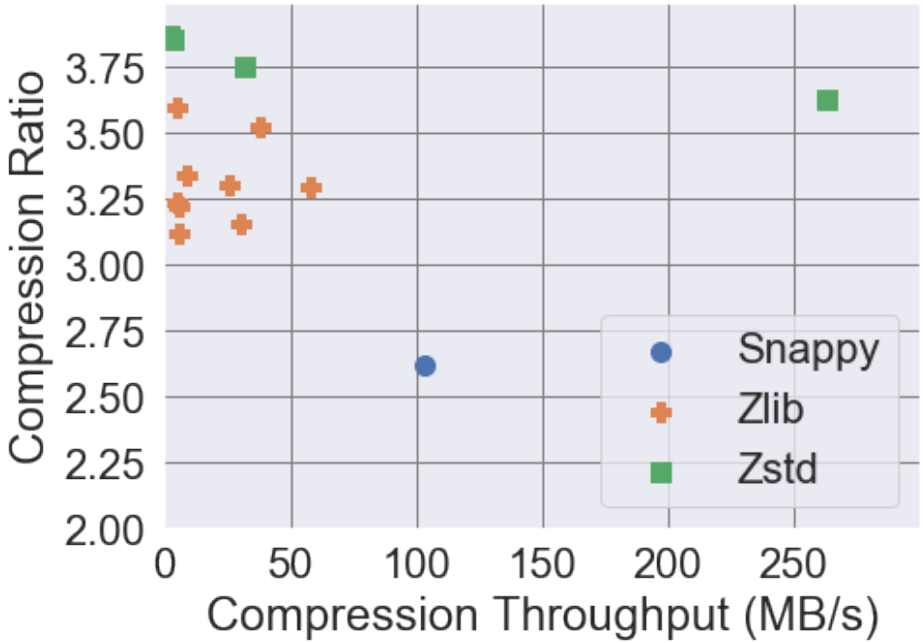
# Compression and Decompression Throughput

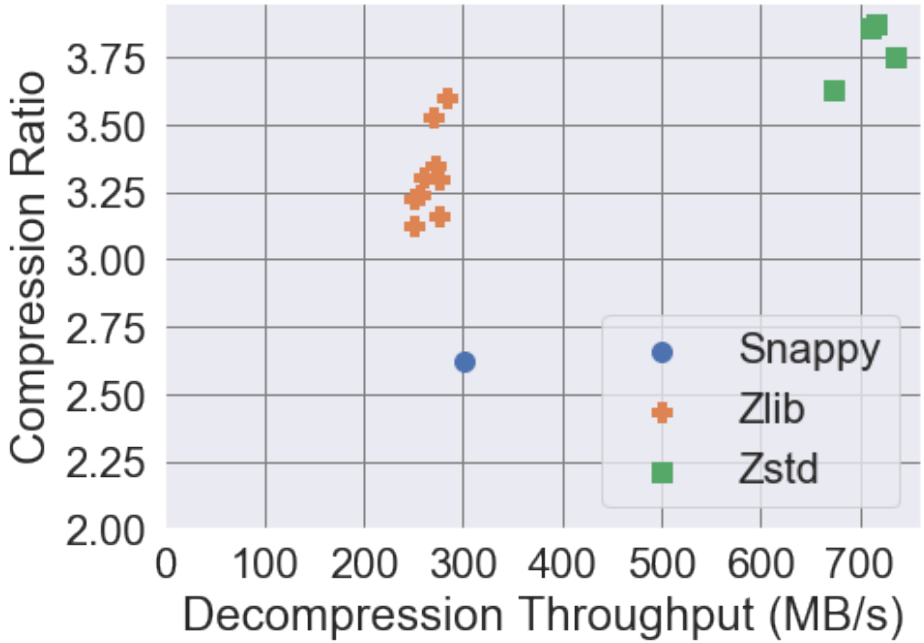



Each dot represents a different configuration for each algorithm.

# Compression and Decompression Throughput




Each dot represents a different configuration for each algorithm.

## Conclusion

Decompression performance is constant regardless of compression settings.

# 2 Selected Strategies

ROOT Configuration Changes

Extracted Branch-Level Storage

# 2 Selected Strategies

## ROOT Configuration Changes

- Switch compression algorithms to `zstd`, level=9
- Add delta encoding and float splitting to ROOT library
- Minimal computational overhead

## Extracted Branch-Level Storage

## ROOT Configuration Changes

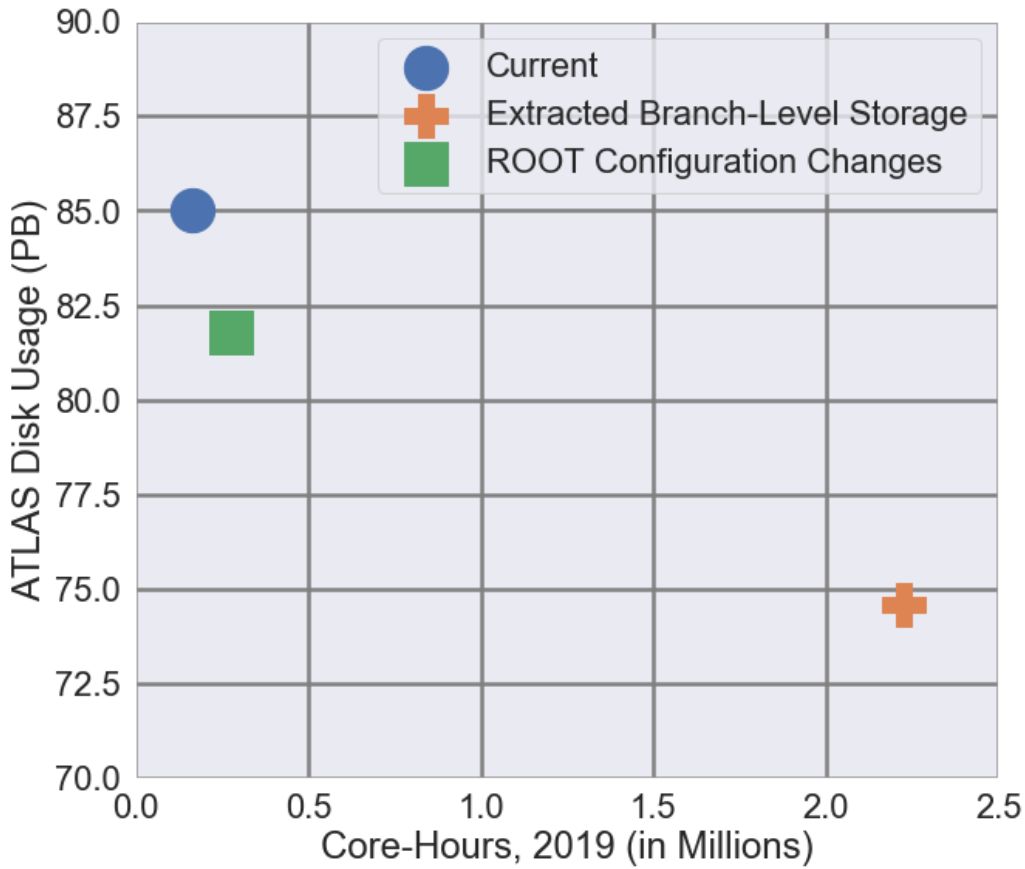
- Switch compression algorithms to `zstd`, level=9
- Add delta encoding and float splitting to ROOT library
- Minimal computational overhead

## Extracted Branch-Level Storage

- Compress and store in aggregate blocks
- Use `zstd`, level=22 and other techniques
- Need additional cores to handle reads and writes

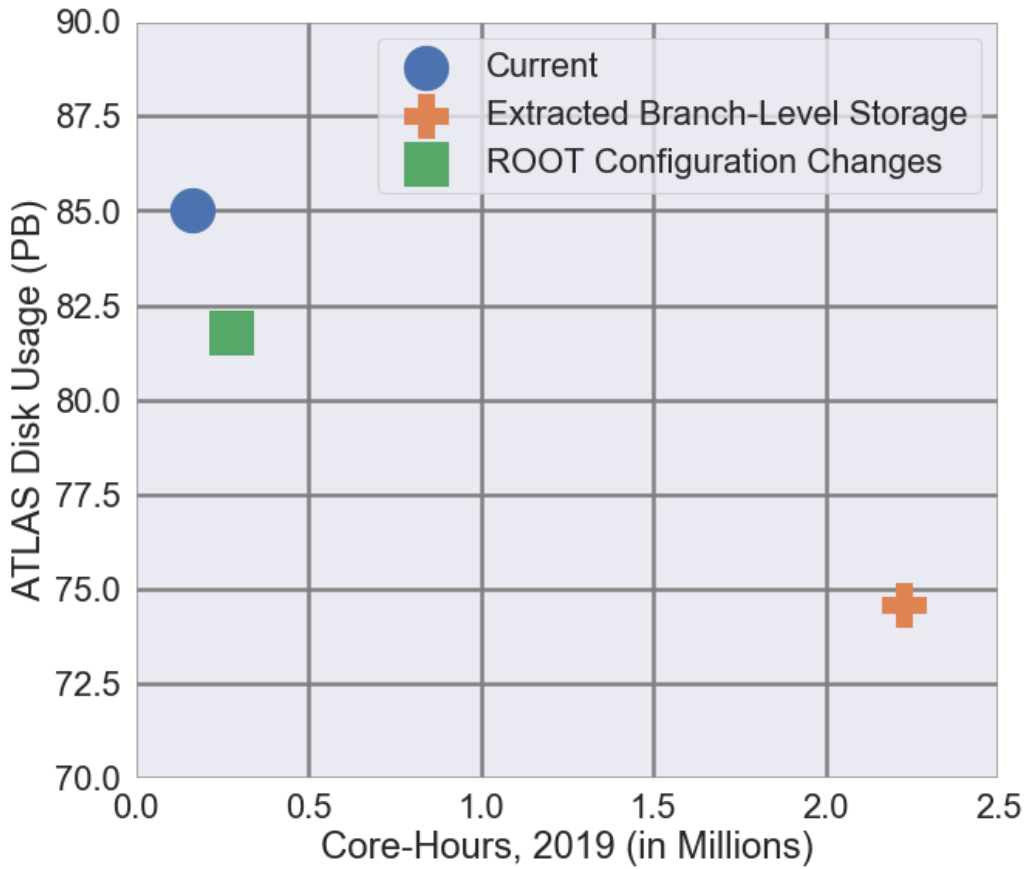
# Saving Storage Space (2019)

- Save up to **12 PB** (15%)



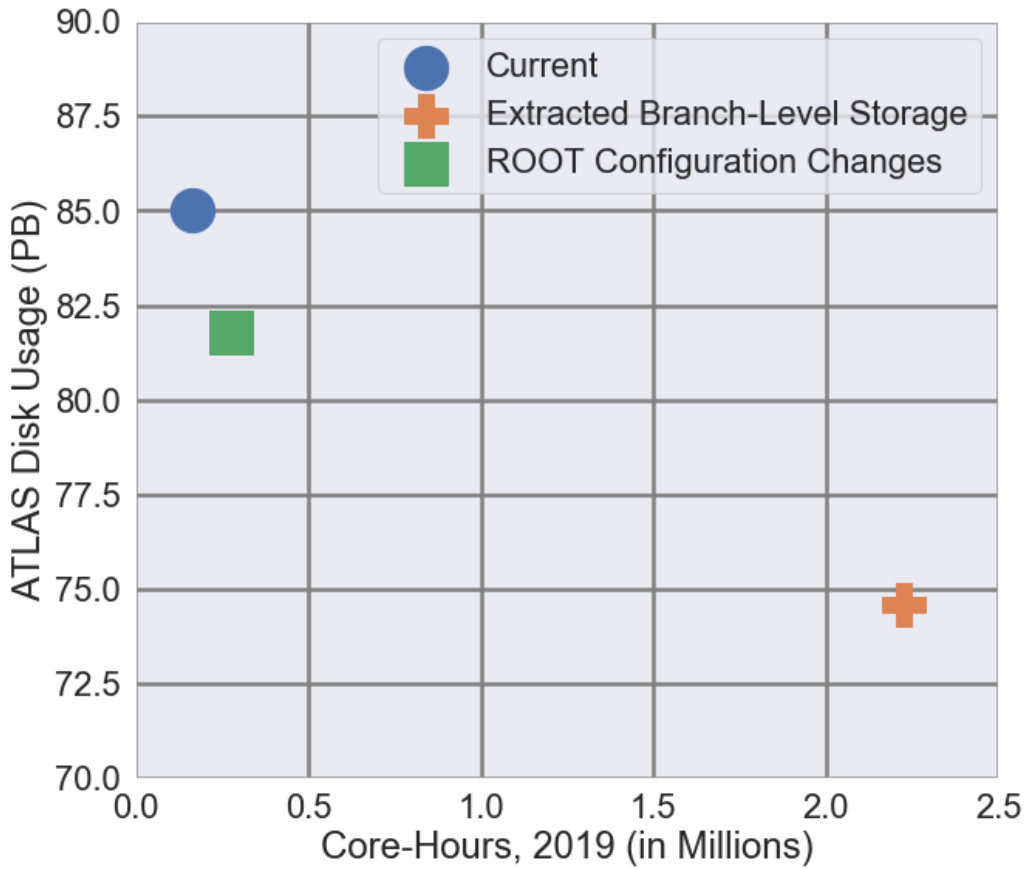
# Saving Storage Space (2019)

- Save up to **12 PB** (15%)
- 2.5M core-hours $<$ 1% of ATLAS 2019 usage

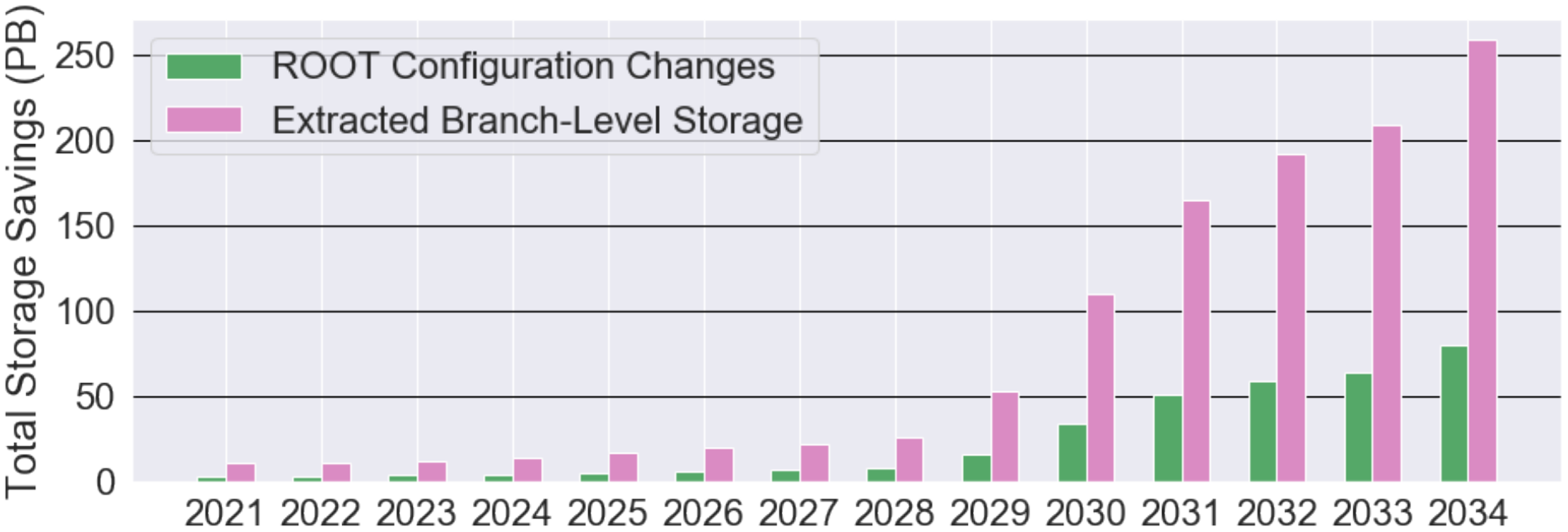


# Saving Storage Space (2019)

- Save up to **12 PB** (15%)
- 2.5M core-hours $<$ 1% of ATLAS 2019 usage
- Exchange storage size for computation
- Many future areas for exploration

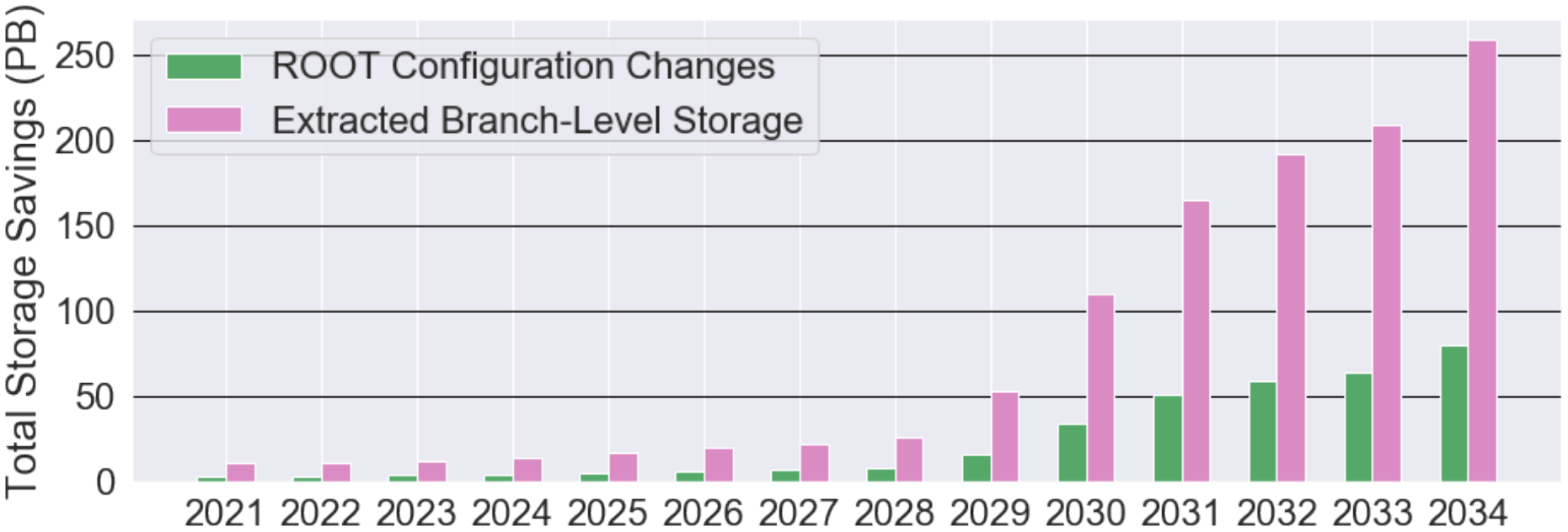


# Saving Storage Space (Projected)



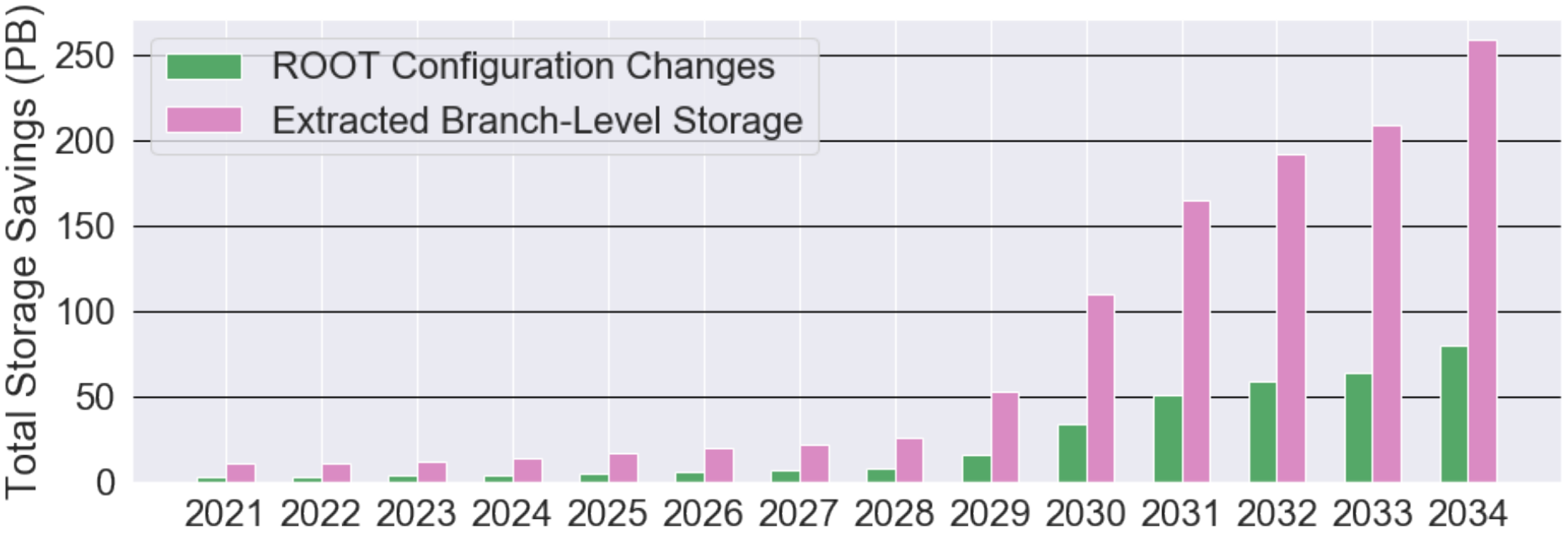

- Total savings of more than 250 PB of data in 2034

# Saving Storage Space (Projected)




- Total savings of more than 250 PB of data in 2034
- Even with CPU costs, still represents savings of millions of dollars

# Saving Storage Space (Projected)




- Total savings of more than 250 PB of data in 2034
- Even with CPU costs, still represents savings of millions of dollars
- Better savings if analysis designed to work outside of ROOT

# Related Work

# Related Work

# Related Work

- ROOT framework
  - Configuration selection for workload (Shadura 2019, ATLAS 2020)
    Speed up R/W times by $> 2$x, but limited to working within ROOT; views compression as black-box

# Related Work

- ROOT framework
  - Configuration selection for workload (Shadura 2019, ATLAS 2020)
    Speed up R/W times by $> 2$x, but limited to working within ROOT; views compression as black-box
- Lossy approaches and filtering
  - Lossy floating point and autoencoder compression (Mete 2020, Heinrich 2019)
    Necessarily loses information, so requires scientific buy-in; can be combined with our approach

# Related Work

- ROOT framework
  - Configuration selection for workload (Shadura 2019, ATLAS 2020)
    Speed up R/W times by $> 2$x, but limited to working within ROOT; views compression as black-box
- Lossy approaches and filtering
  - Lossy floating point and autoencoder compression (Mete 2020, Heinrich 2019)
    Necessarily loses information, so requires scientific buy-in; can be combined with our approach
- Domain specific compression (Non-HEP)
  - Learned compression for objects (Chen 2017, Sanchez 2019)
    Can be integrated into our approach to create specific compression techniques for any object

# Related Work

- ROOT framework
  - Configuration selection for workload (Shadura 2019, ATLAS 2020)
    Speed up R/W times by $> 2$x, but limited to working within ROOT; views compression as black-box
- Lossy approaches and filtering
  - Lossy floating point and autoencoder compression (Mete 2020, Heinrich 2019)
    Necessarily loses information, so requires scientific buy-in; can be combined with our approach
- Domain specific compression (Non-HEP)
  - Learned compression for objects (Chen 2017, Sanchez 2019)
    Can be integrated into our approach to create specific compression techniques for any object

# Summary and Future Work

# Summary of Contributions

# Summary of Contributions

1. Evaluated performance of compression algorithms and strategies on HEP data

# Summary of Contributions

1. Evaluated performance of compression algorithms and strategies on HEP data

2. Designed strategies to improve compression ratio for data storage

# Summary of Contributions

1. Evaluated performance of compression algorithms and strategies on HEP data
2. Designed strategies to improve compression ratio for data storage
3. Achieved data storage reduction of 15% on ATLAS DAOD (82 GB $\rightarrow$70 GB)

# Summary of Contributions

1. Evaluated performance of compression algorithms and strategies on HEP data
2. Designed strategies to improve compression ratio for data storage
3. Achieved data storage reduction of 15% on ATLAS DAOD (82 GB $\rightarrow$70 GB)
4. Modeled scale-up for ATLAS (Save $>$250 PB by 2035)

# Future Work

1. Can strategies like float splitting be generated automatically for any datatype/distribution?

2. Could data access patterns inform compression strategy choices?

3. Can storage acceleration be used to offload computational cost to accelerators?

# Acknowledgements

Prof. Andrew A. Chien

Chen Zou

Ilija Vukotic

Prof. Rob Gardner

Large-Scale Systems Group

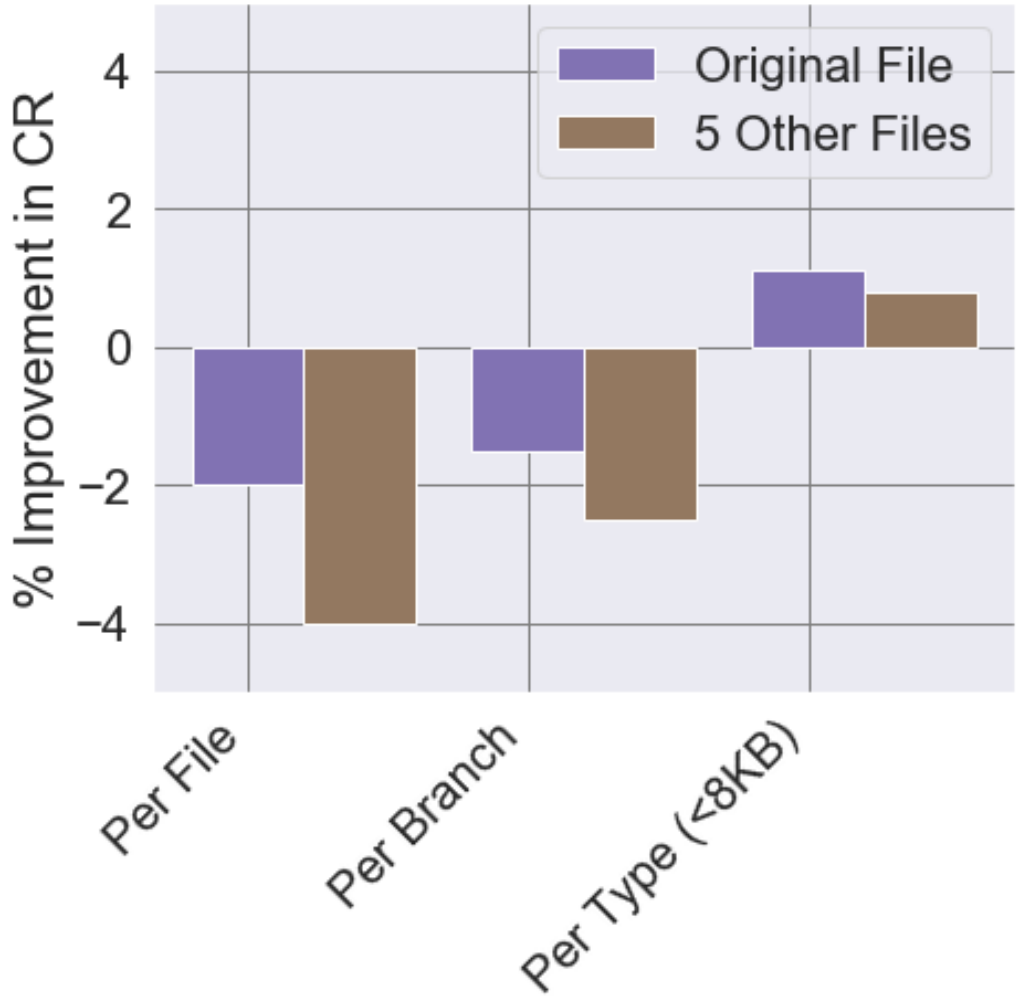
MANIAC Lab

# Questions?

# Backup Slides

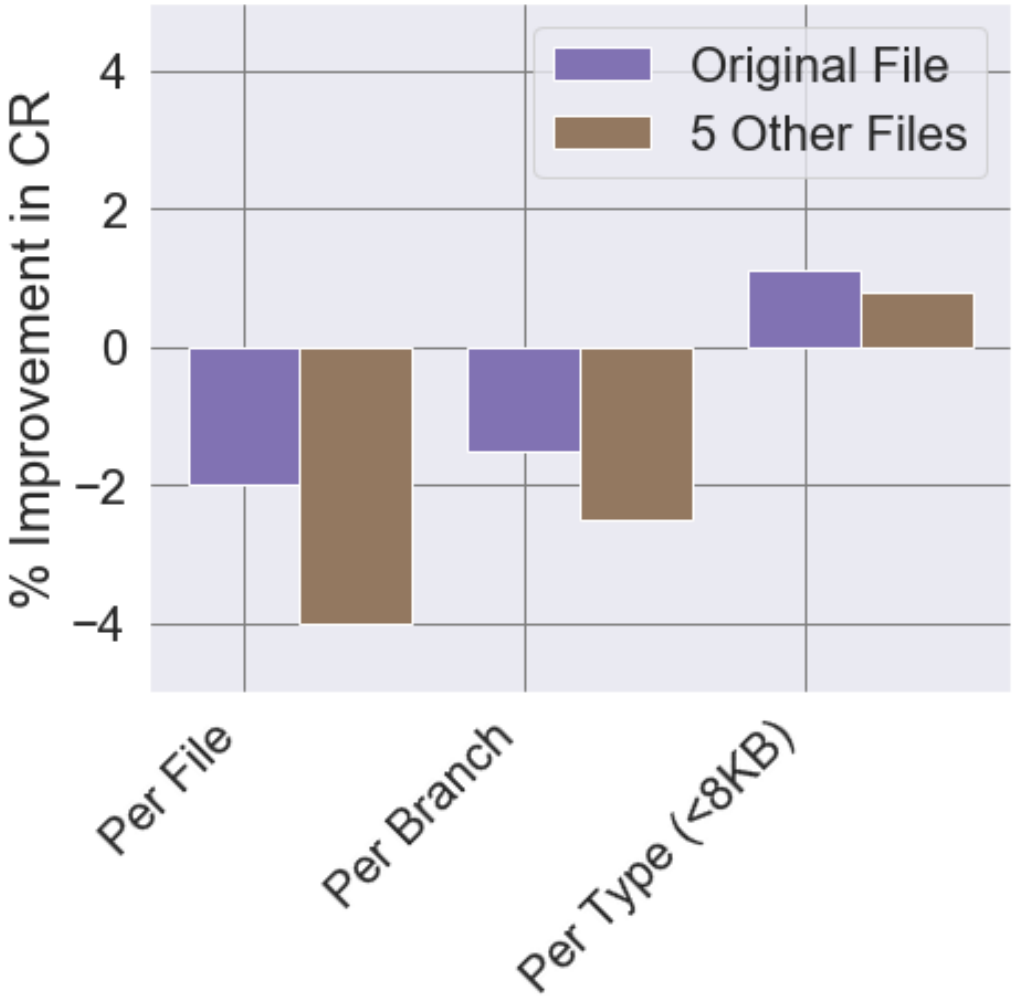
# Pretraining Dictionaries

- Train 1 dictionary per datatype with `zstd`

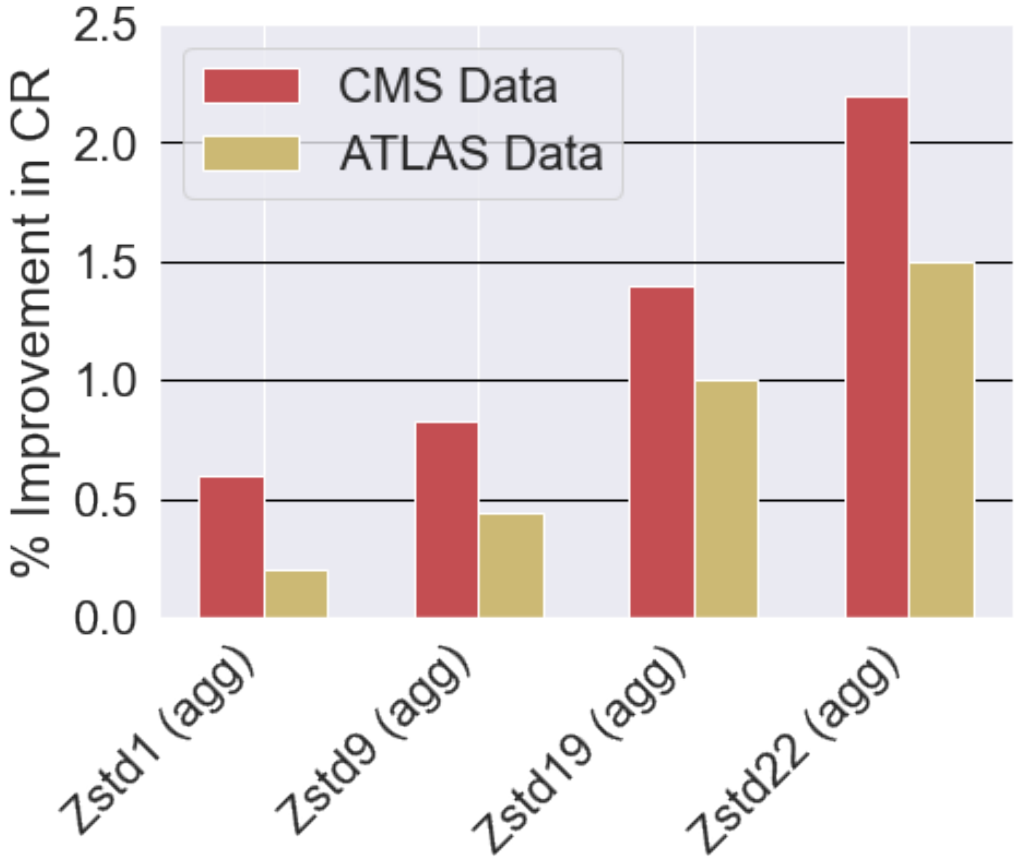


# Pretraining Dictionaries

- Train 1 dictionary per datatype with `zstd`
- Dictionary training does not universally improve compression ratio
- However, we realize a 1% improvement in CR when pretraining on small branches ($< 8$KB)



# Aggregate Strategies (Across Files)

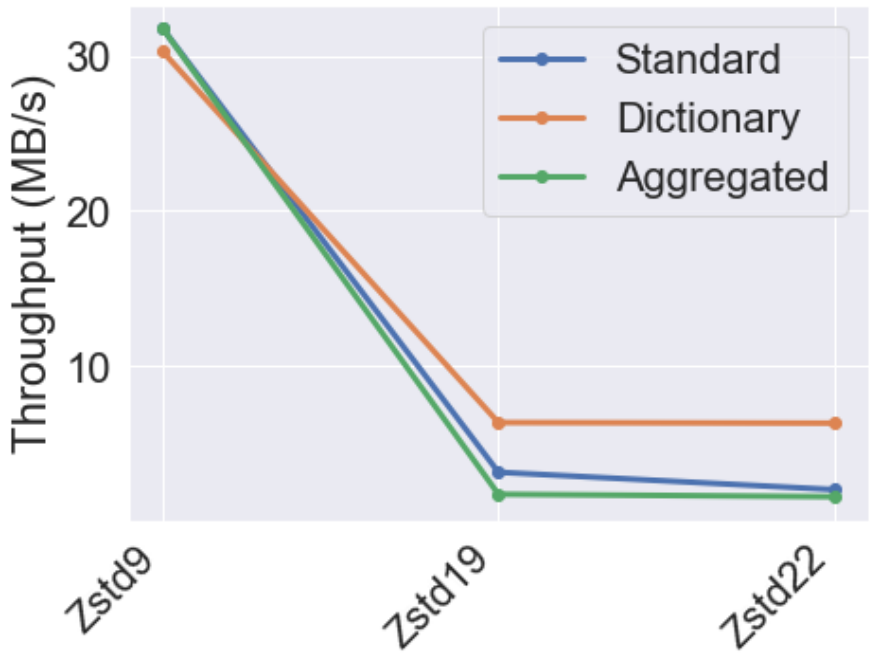



## Key Insight

Aggregating across files can improve compression ratio by up to 2%.

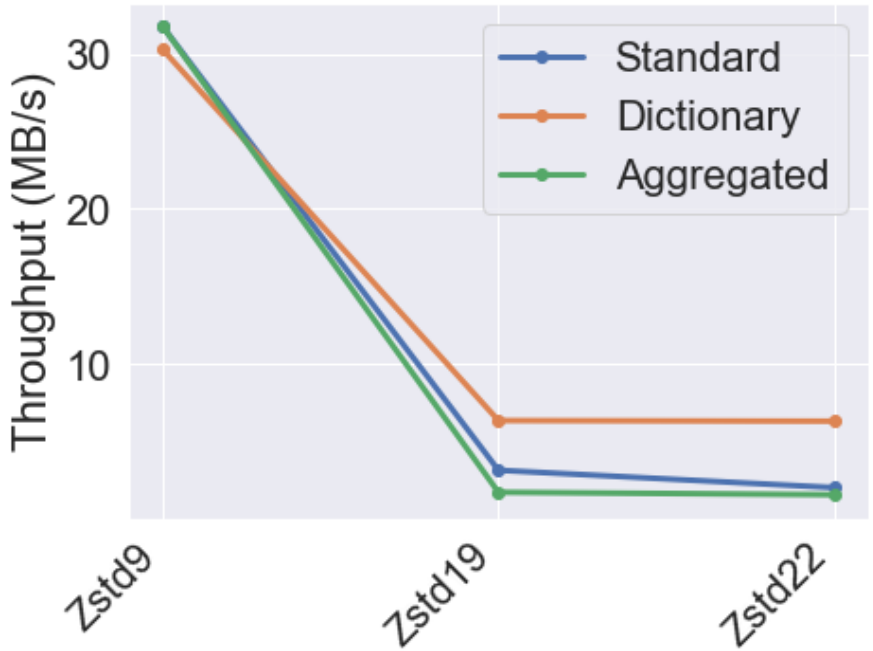
# Other Techniques Can Improve Throughput

- Pretraining improves compression throughput by **3x**



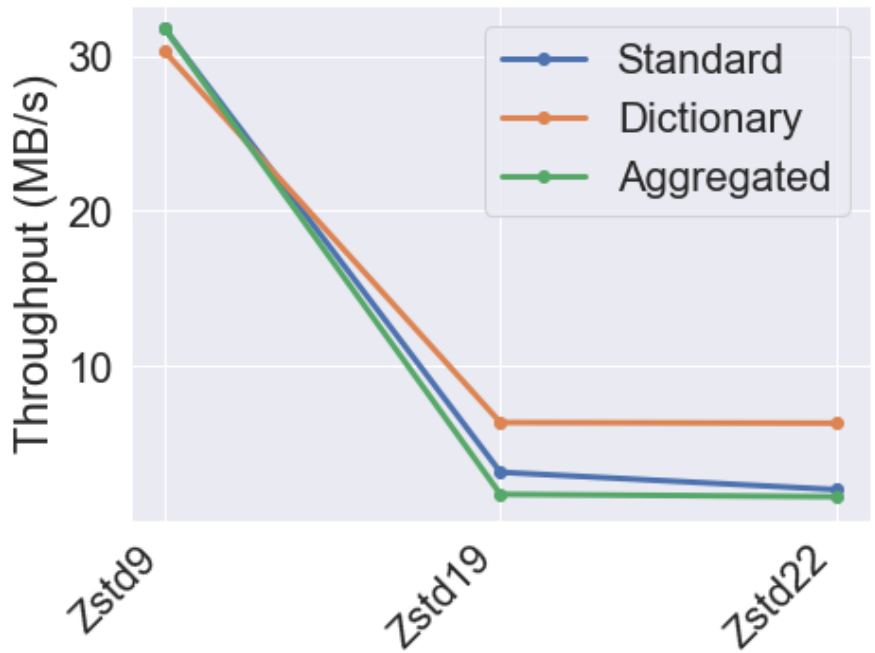
# Other Techniques Can Improve Throughput

- Pretraining improves compression throughput by **3x**
- Delta encoding and other passes run at $> 300$ MB/s

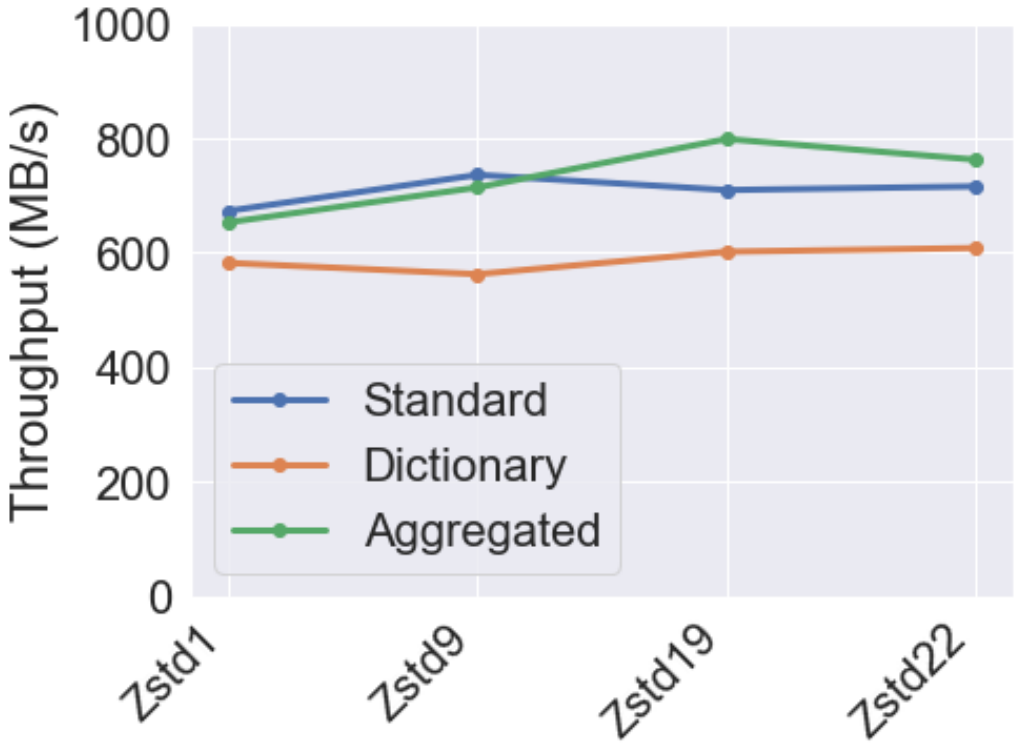


# Other Techniques Can Improve Throughput

- Pretraining improves compression throughput by **3x**
- Delta encoding and other passes run at $> 300$ MB/s
- Creating large baskets is not expensive (100 MB/s)



# Decompression Throughput for Zstd Variations




## Key Insight

`zstd` with aggregation and dictionary do not have a significant effect on decompression throughput, all still >575 MB/s.