# DIANA Contributions Update

Brian Bockelman
Including work from Jim Pivarski, Oksana Shadura, and Zhe Zhang

# DIANA Contributions (Since July)

- Since the F2F meeting, DIANA contributions have focused on the following areas:
  - **Parallelism**: parallel, asynchronous unzipping of baskets (utilizing TBB).
  - **Compression**: LZ4, removing redundant information from data formats.
  - **Bulk IO**: Shown to be much faster for small events; working to broaden the impact.
- I'll give some highlights in each area and discuss planned work.

# Parallelism:
# Parallel, Asynchronous Unzipping

- Currently, IMT-enabled reading of branches causes all baskets in all active branches in the current event cluster to be decompressed before control is returned to user thread.
    - This is a **synchronous serialization point**: user must wait for all baskets to decompress, including tails.
    - Unless user utilizes TDataFrame (*which they should!*), it's likely the user thread is single-threaded: many idle cores between IO calls!
- Zhe has been making this activity **asynchronous.** Control is returned to the user thread immediately and separate TBB tasks are launched to do decompression. User thread is only blocked when data is needed.
    - Builds on top of old pthread-based parallel interface but implementation instead invokes the ROOT TBB wrapper classes.
- See: https://github.com/root-project/root/pull/1010
    - Performance beats current IMT mode - as it should. Tested on a large range of branch layouts.
    - Ready to be reviewed more in-depth and merged!

# Compression: LZ4

- Backported to all active release series (5.34, 6.08, 6.10, & in master).
- See Jim's presentation for detailed numbers.
  - See also: https://indico.cern.ch/event/567550/contributions/2627167/
  - **Short version**: for reading, nearly the same performance as uncompressed files. File-size penalty versus default zlib varies (depends highly on contents!), but is around 15%.
- **Decision point**: change to default "today" or wait until after 6.12 has branched?

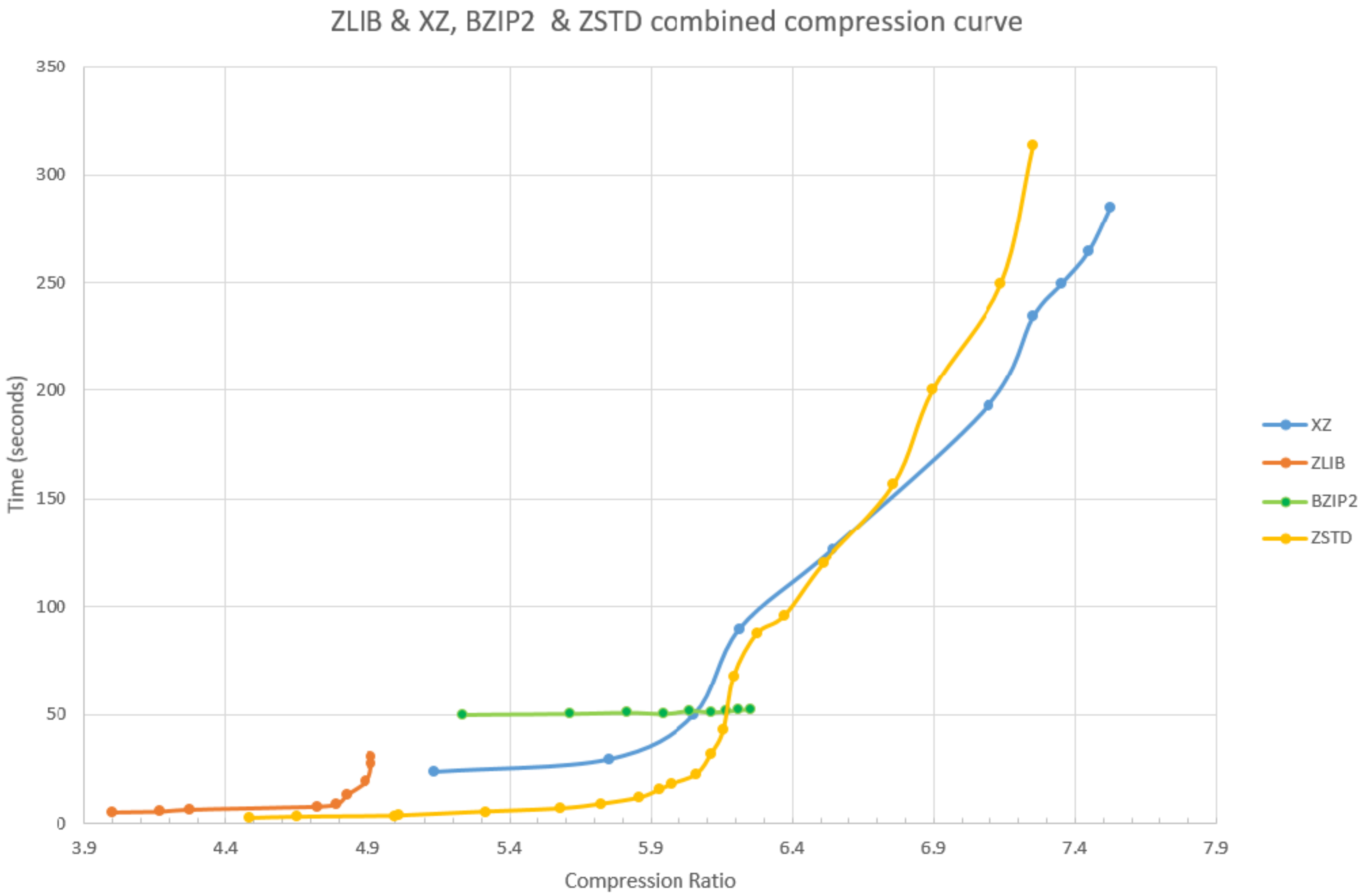
# ZLIB Updates

- We have continued to make progress on the goal of getting CloudFlare's zlib speed improvements in ROOT.
    - Approximate improvement is 20% in decompression speeds and 2x-4x for compression (zlib-6 vs zlib-9).
- Oksana has gotten their patches building inside ROOT *and* make sure there aren't regressions on other platforms.
    - CloudFlare tailored them to only work on new x86-64 cores…
- This has been a bit stuck due to other zlib-related cleanups:
    - Unit test failures if `-ffast-math` is enabled.
    - Getting ROOT to use only *one* version of zlib. Currently can be up to 3!
- CMS has been using these quite happily: default changed from zlib-7 to zlib-9. Smaller files and less time spent in compression. [Note: most data by volume probably still in LZMA.]
- See: https://github.com/root-project/root/pull/956

# ZSTD

- Yet another compression algorithm? **Why would you do that, Brian?**
  - **Answer #1**: ZSTD is flexible and fast. Depending on target level, competitive with LZ4, LZMA, and ZLIB. Better than ZLIB (compression ratio / speed) across the board. Not as good as LZ4 and LZMA for at the extremes.
- See: http://facebook.github.io/zstd/

**Source: https://clearlinux.org/blogs/linux-os-data-compression-options-comparing-behavior**

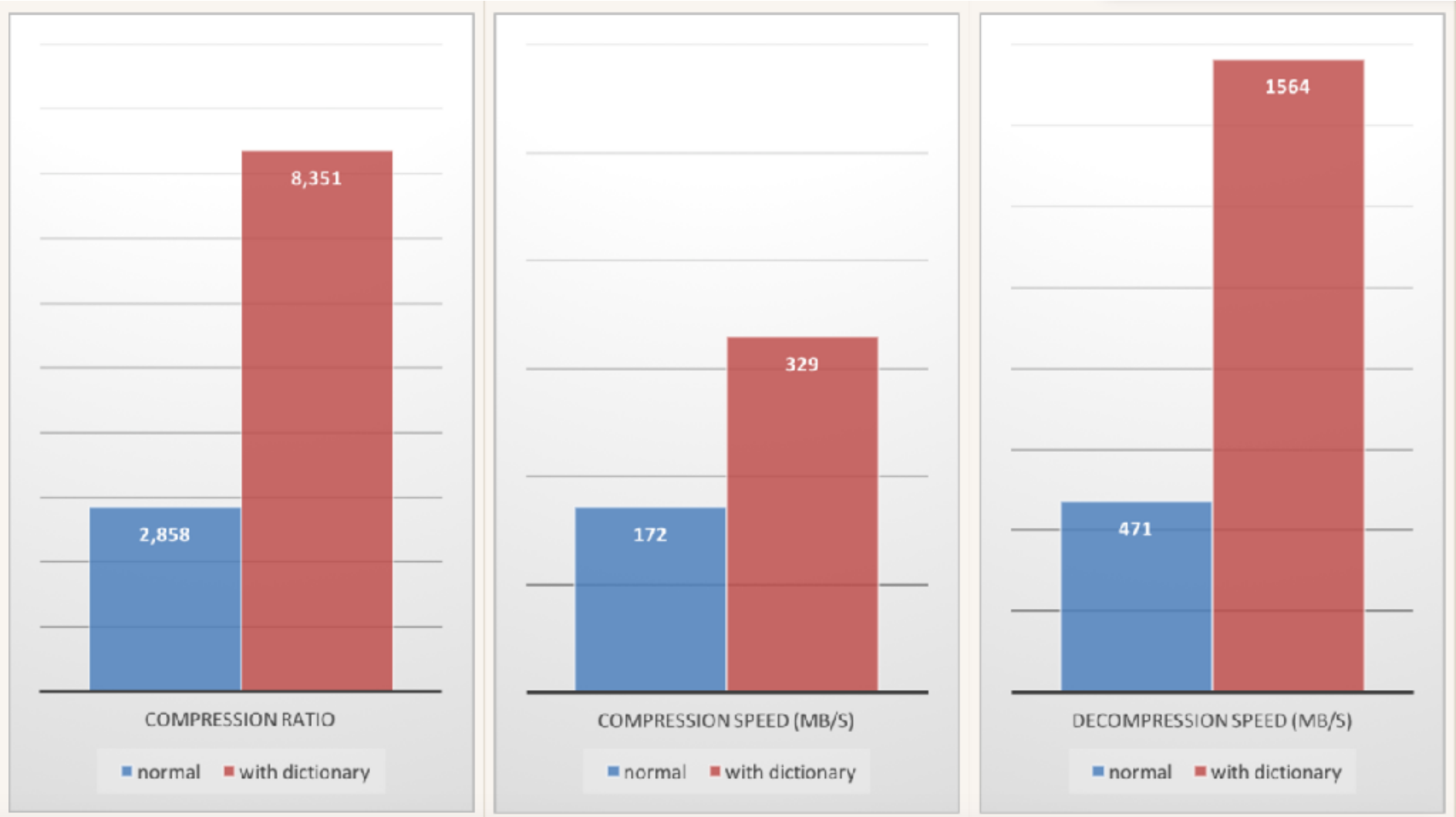


# ZSTD - Answer #2

- ZSTD is also interesting because it has a rich API for generating and using compression dictionaries.
  - Facebook developers report massive speedups and compression ratio improvements when using dictionaries (almost a 3x improvement in compression ratio!) on a corpus of 10,000 entries of 1KB each.
- If we can get *anything near that*, then it would be a huge improvement for ROOT.
  - **Idea**: after the first event cluster, analyze the buffer and write out a separate compression dictionary.
  - No clue how much of Facebook's success can be repeated in ROOT, but appears worth investigating this winter.

# ZSTD - With Dictionaries

**Source: http://facebook.github.io/zstd/**



# Compression

- We have found a few places in the file format where we can skip writing out redundant information.
    - Some of the redundancies compress well, meaning the savings is in CPU time and memory use.
    - Some compressed poorly, meaning the savings is in output file size.
- Have an almost-merged PR for removing redundancies from entry offset arrays.
    - Will be a few follow-ups to improve the range of classes where this technique is applicable.
    - (Again, more from Jim / Oksana later today)
- There are additional savings to be had in removing redundant class version information.
- There are some degenerate cases where buffer offset arrays can be skipped without forward-compatibility breaks.

# Forward Compatibility Breaks

- The entry offset work motivated us to finally figure out a mechanism for **cleanly introducing forward-compatibility break**.
- Solving this long-standing problem is a prerequisite for more innovation at the file-format level.
- **NOTE**: intent is that these features are disabled by default until ROOT7.

# Bulk IO

- Since the last F2F, the bulk IO:
  - Matured enough to build two high-level interfaces (Python/numpy and TTreeReader-like).
  - Got enough functionality to do realistic performance comparisons.
  - Got into a reviewable state and put in as a PR.
- First round of review done: quite a few changes to do (but very good suggestions from Philippe!), but fundamental idea remains solid.
  - Goal: have this merged this calendar year (but likely not in time for 6.12)

# Bulk IO-Inspired

- Bulk IO has inspired two sub-projects in ROOT IO:
  - **One-basket-per-cluster** mode: Having multiple baskets per event cluster triggers significant special-case code. This extra overhead is noticeable in the bulk IO performance tests. This mode will cause buffer memory to grow until an entire cluster is serialized.
    - This branch needs a few more tests and documentation, then is ready to be merged.
  - Fully-split mode for `std::vector` of primitive types. Currently, `std::vector<int>` is never-split, causing performance penalties when used from bulk IO.
    - Work not started.

# Bulk IO - Other

- Would like to use the TDataSource facility with bulk IO, potentially turbocharging TDataFrame use.
  - Have some concern that TDataFrame is not yet fast enough for bulk IO to matter.
  - What's the best performance test along these lines? **Future investigation needed**!
- With bulk IO and modern storage technology (NVMe, Intel X-Point), we may finally benefit from utilizing mmap to minimize latency for reading files: **future investigation needed**!

# Preferred/Predicted Timelines for Merging

- #1003 - skip writing basket offset arrays.
- #1010 - Parallel, asynchronous unzipping.
- #240 - Miss cache. Improves behavior when accessing infrequently-used branches.

**6.12 forks**

- #956 - Improved zlib. May depend on #1149 (using one, consistent version of zlib throughout ROOT. See ROOT-8839)?
- #943 - Bulk IO. See prior discussion. Needs to rework to avoid interface changes for TBufferFile.
- #774 - “one basket per cluster” mode.
- Note this isn’t in priority order - some high-priority items (bulk IO) are likely coming later due to the number of interface changes.

Questions?