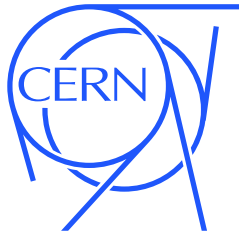


A quantitative review of data formats for HEP

Jakob Blomer, CERN

Many thanks to Philippe Canal and Axel Naumann!

ACAT 2017, Seattle

August 24th, 2017

Assumptions in this Talk

Focus on data formats for the "last mile" of publication creation

- Data sets are relatively small
  - $\mathcal{O}(10^7)$ events
  - Tens to hundreds of properties per event
  - Volume: gigabytes to terabytes
- Little data lock-in: final data sets can be reproduced
  e. g. starting from CMS MINIAOD, ATLAS Derived xAOD
- Processing tends to escape central computing workflows and resources
- Number of reads $\gg$ number of writes
- I/O bound: little calculation

The Role of Data Formats

What we want from the data format

1. Fast turn-around, e. g.
   "Let me quickly plot. . . "
   - Tuning cuts (partial reading of the data set)
   - Feeding into machine-learning framework
     (full reading of the data set)
2. Integration, unleashing the data
   - Libraries for C++, Python
   - Machine-learning frameworks (e. g. TMVA, TensorFlow)
   - Big Data schedulers (e. g. Spark)
   - Analytic tool kits (e. g. ROOT, R, SciPy)

A Menu of Data Formats

ROOT:
- Stream of serialized C++ objects in columnar layout

Protobuf:
- Not a file format per se but (de)-serialization of small records
- Possible file format: `schema || [size record-blob]*`

SQlite:
- SQL database in a file

HDF5:
- Hierarchical "data sets" that each contain a "data space" (*n*-dimensional array)
- Popular in the HPC universe, integrated with MPI-I/O
- There are HEP machine-learning data sets in HDF5

Parquet:
- From the Apache Hadoop/Spark universe
- Column-wise binary storage

Avro:
- From the Apache Hadoop/Spark universe
- JSON schema, row-wise binary storage

Kudu:
- C++ native implementation of a format similar to Parquet
- Not (yet?) available as library, only client-server APIs

Functional Data Format Properties

	ROOT	PB	SQlite	HDF5	Parquet	Avro
Well-defined encoding	✓	✓	✓	✓	✓	✓
C/C++ Library	✓	✓	✓	✓	✓	✓
Self-describing	✓	↯	✓	✓	✓	✓
Nested types	✓	✓	?	?	✓	✓
Columnar layout	✓	↯	↯	?	✓	↯
Compression	✓	✓	↯	?	✓	✓
Schema evolution	✓	↯	✓	↯	?	?

✓ = supported
↯ = unsupported
? = difficult / unclear

"Fruit Fly" Data Set: The LHCb OpenData Sample

Starting point: "What if I had my data set in format X?"

Example Analysis

- 8.5 million LHC run 1 events $B \to KKK$ ▸ Link
- Flat n-tuple, 26 branches,
  mostly floating point numbers
- 21 branches needed for the toy analysis
- 2.4 million events can be skipped
  because one of the kaon candidates is flagged
  as a muon
- Toy analysis:
  sum over all branches from non-cut Kaons

```
struct BDecay {
  double h1_px;
  double h2_px;
  double h3_px;
  double h1_py;
  ...
};
```

On the simple end of the spectrum,
helps to understand performance base case

Results

Bit Flips

Outside managed storage systems (e. g. EOS), files are vulnerable to "bit rot".
Without checksums, silently flipped bits can change physics results!

Test

- 500 k events, 60 MB to 100 MB
- Run toy analysis
- 100 trials with one random bit flip

Notes

- Usually: checksumming as a byproduct of compression
- ROOT reports errors but does not crash
- ROOT occasionally runs into infinite loop (under investigation)

Format	Undetected Errors
ROOT (uncompressed)	32 %
ROOT (zlib)	0 %
ROOT (lzma)	0 %
ROOT (lz4)	32 %
sqlite	32 %
HDF5 (column)	39 %
HDF5 (row)	37 %
protobuf (uncompressed)	34 %
protobuf (zlib)	0 %
parquet (uncompressed)	33 %
parquet (zlib)	0 %
avro (uncompressed)	34 %
avro (zlib)	23 %

Bit Flips

Outside managed storage systems (e. g. EOS), files are vulnerable to "bit rot".
Without checksums, silently flipped bits can change physics results!

Test

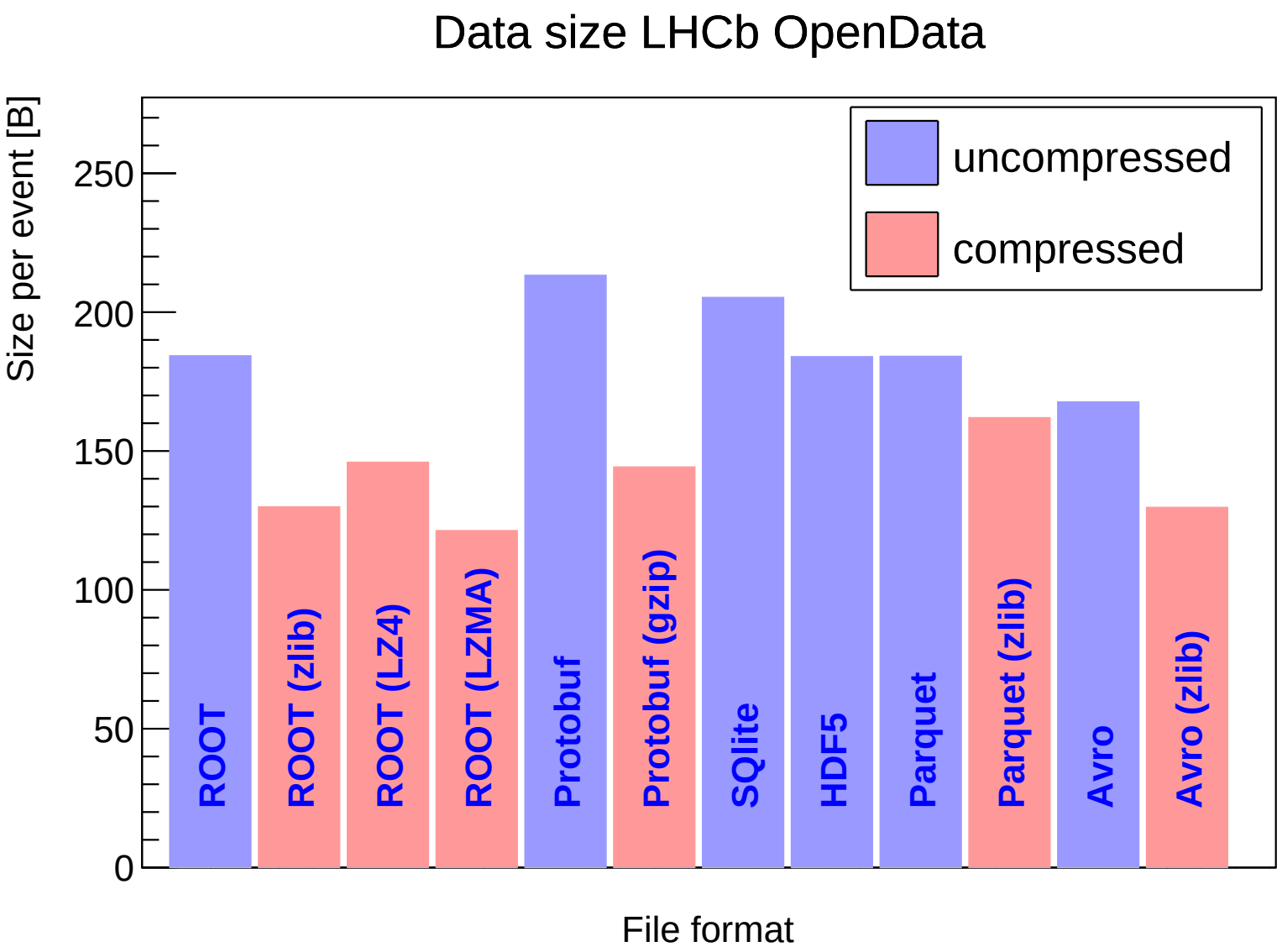
- 500 k events, 60 MB to 100 MB
- Run toy analysis
- 100 trials with one random bit flip

Notes

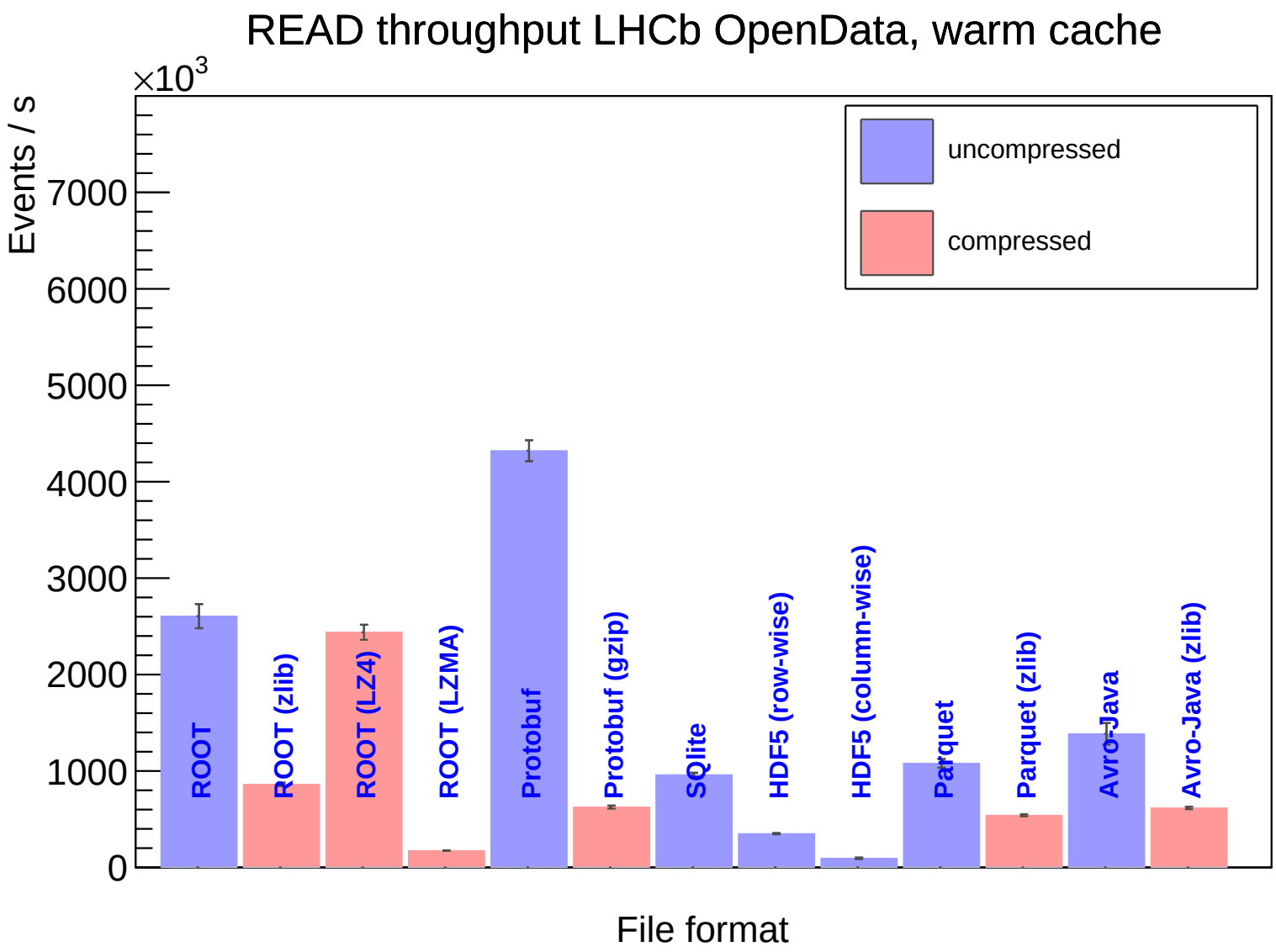
- Usually: checksumming as a byproduct of compression
- ROOT reports errors but does not crash
- ROOT occasionally runs into infinite loop (under investigation)

Format	Undetected Errors
ROOT (uncompressed)	32 %
ROOT (zlib)	0 %
ROOT (lzma)	0 %
ROOT (lz4) under investigation	32 %
sqlite	32 %
HDF5 (column)	39 %
HDF5 (row)	37 %
protobuf (uncompressed)	34 %
protobuf (zlib)	0 %
parquet (uncompressed)	33 %
parquet (zlib)	0 %
avro (uncompressed)	34 %
avro (zlib) checksums turned off	23 %

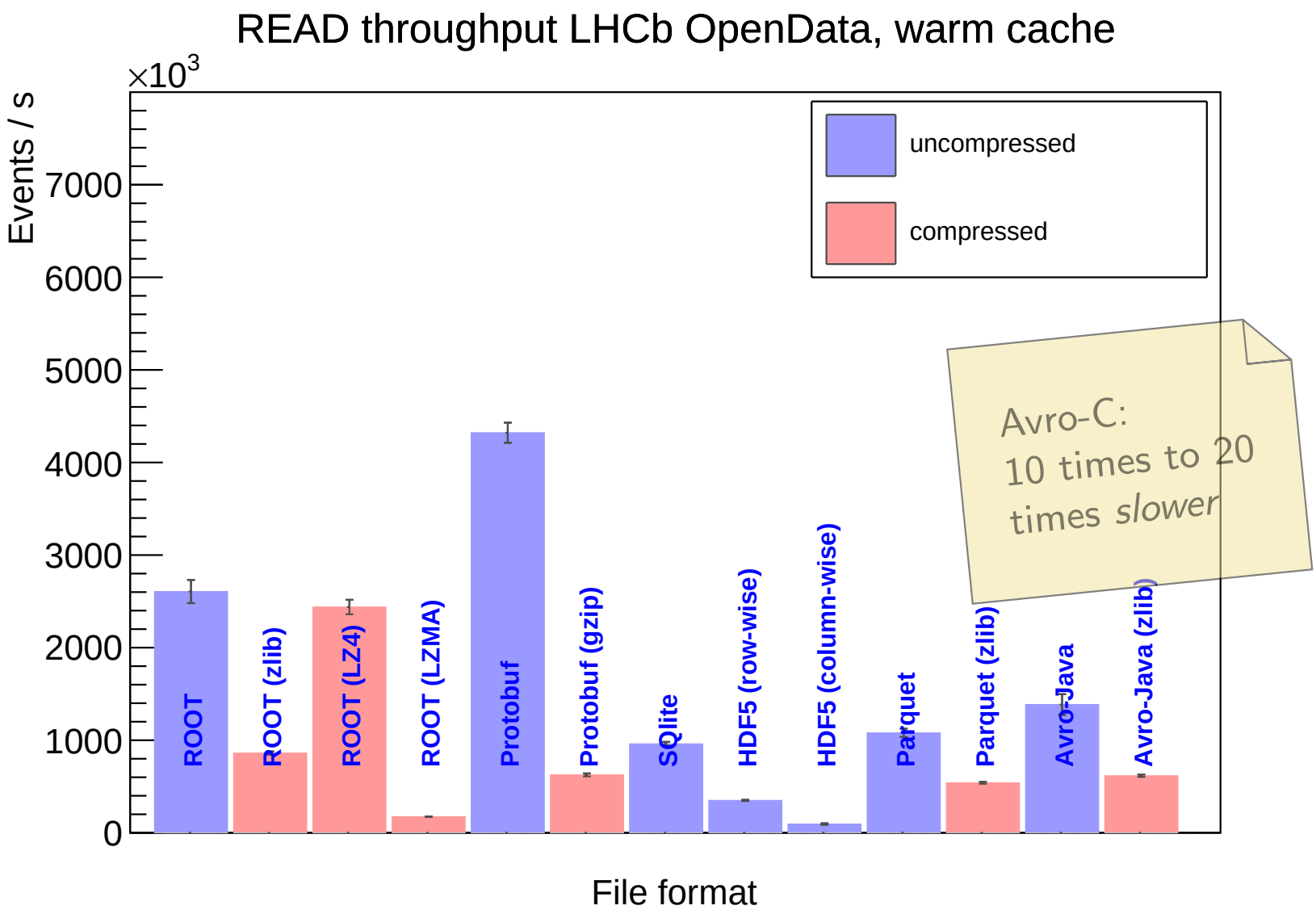
Encoding Efficiency



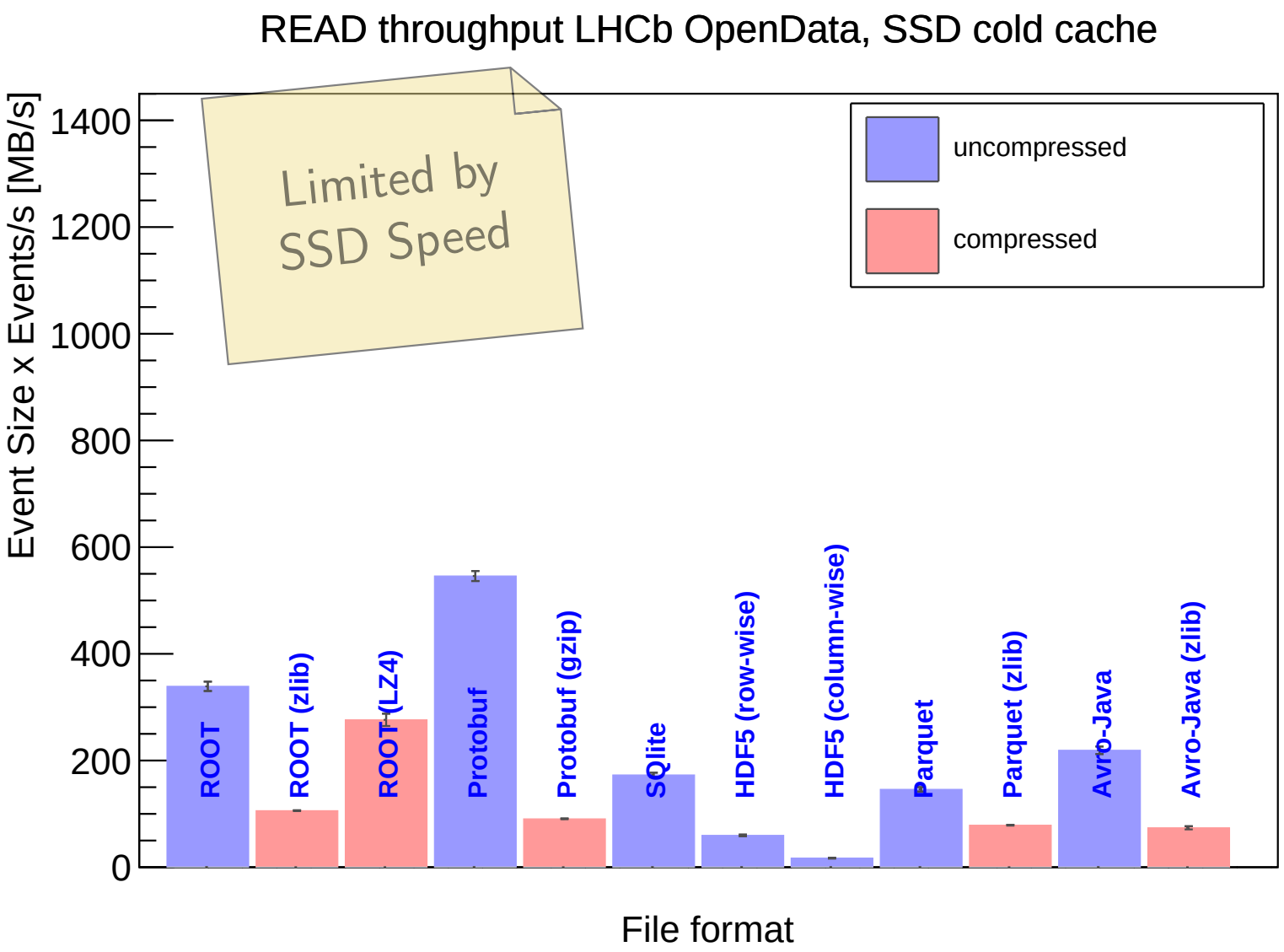
Read Throughput (Memory, Events/s)



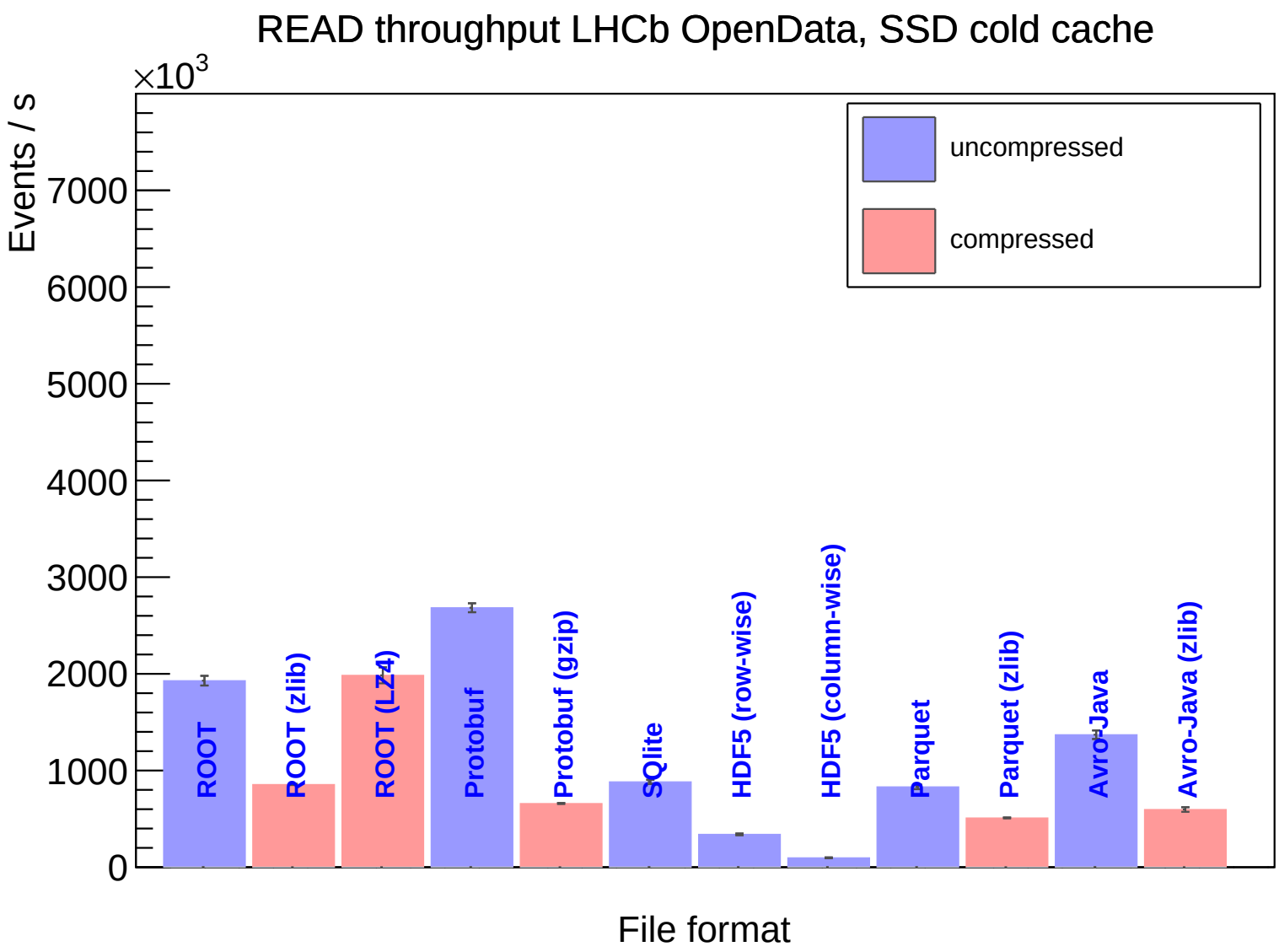
Read Throughput (Memory, Events/s)



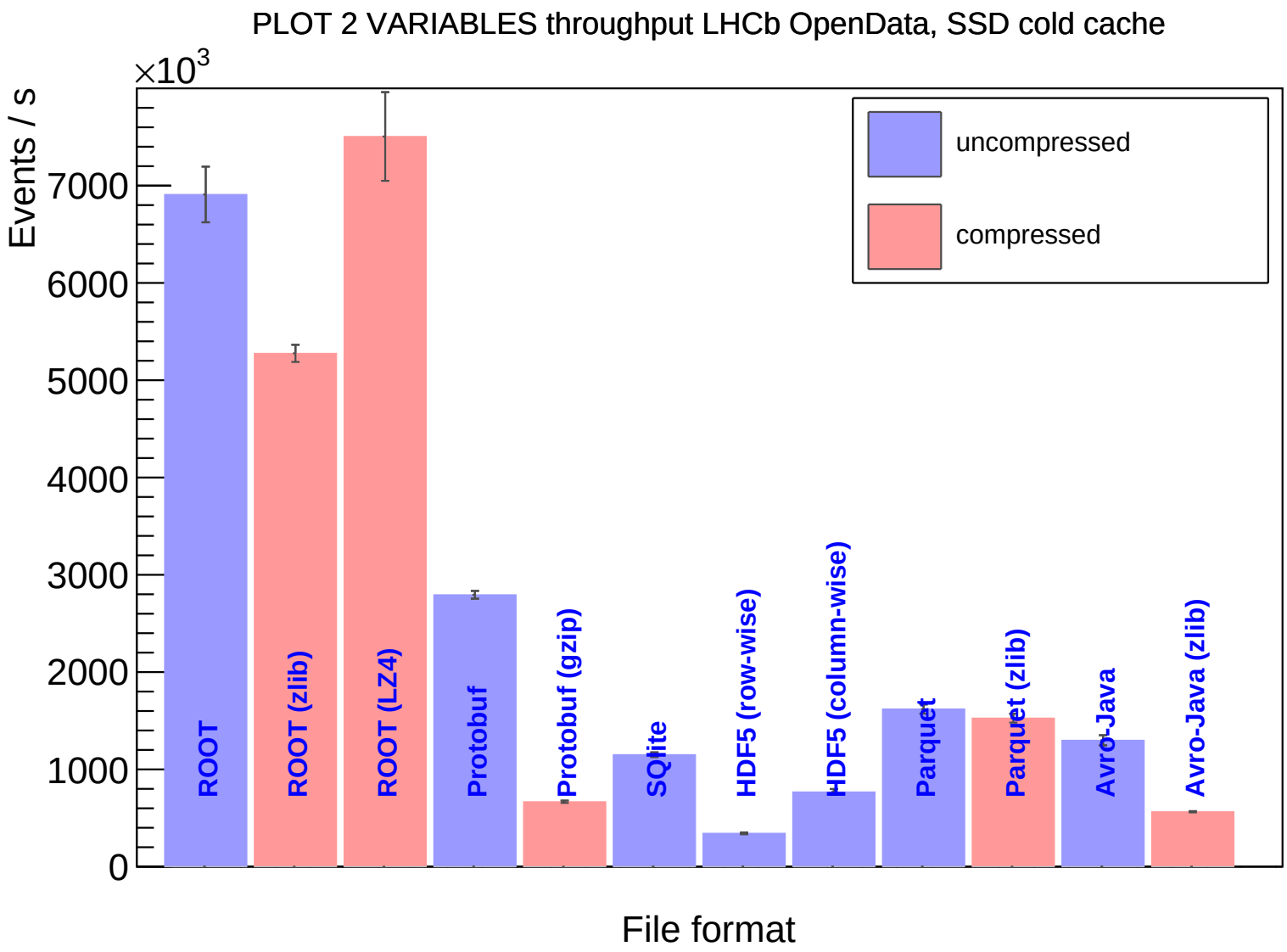
Read Throughput (SSD, MB/s)



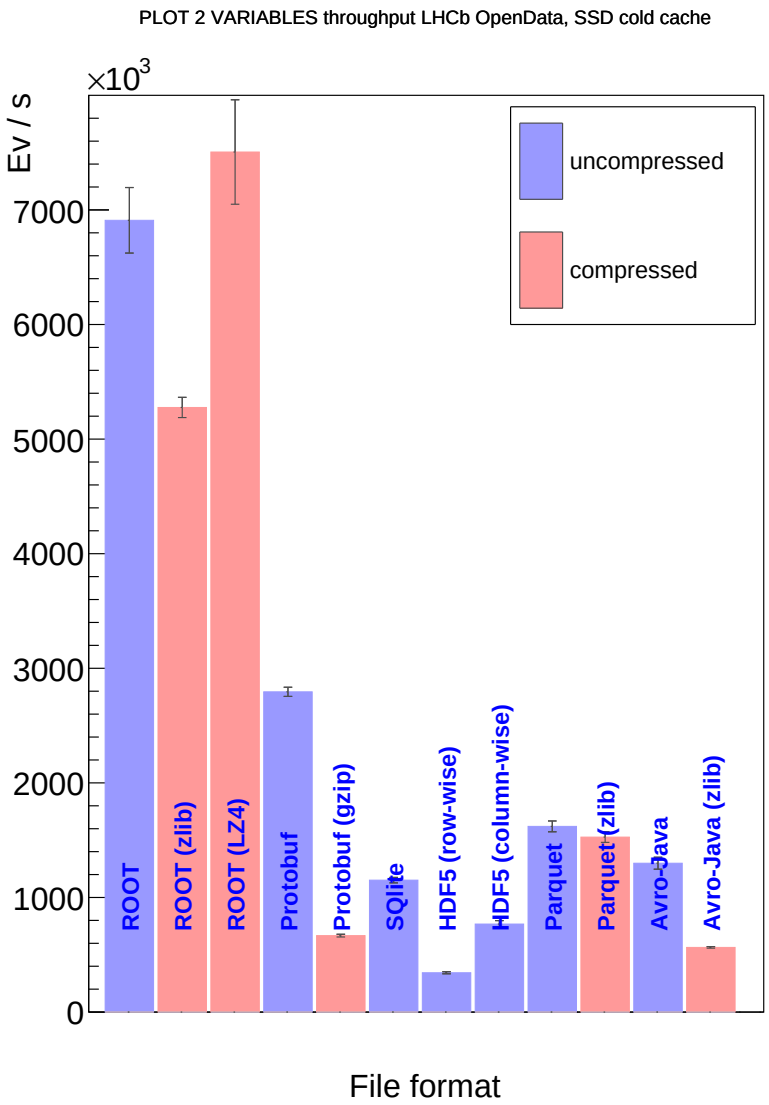
Read Throughput (SSD, Events/s)

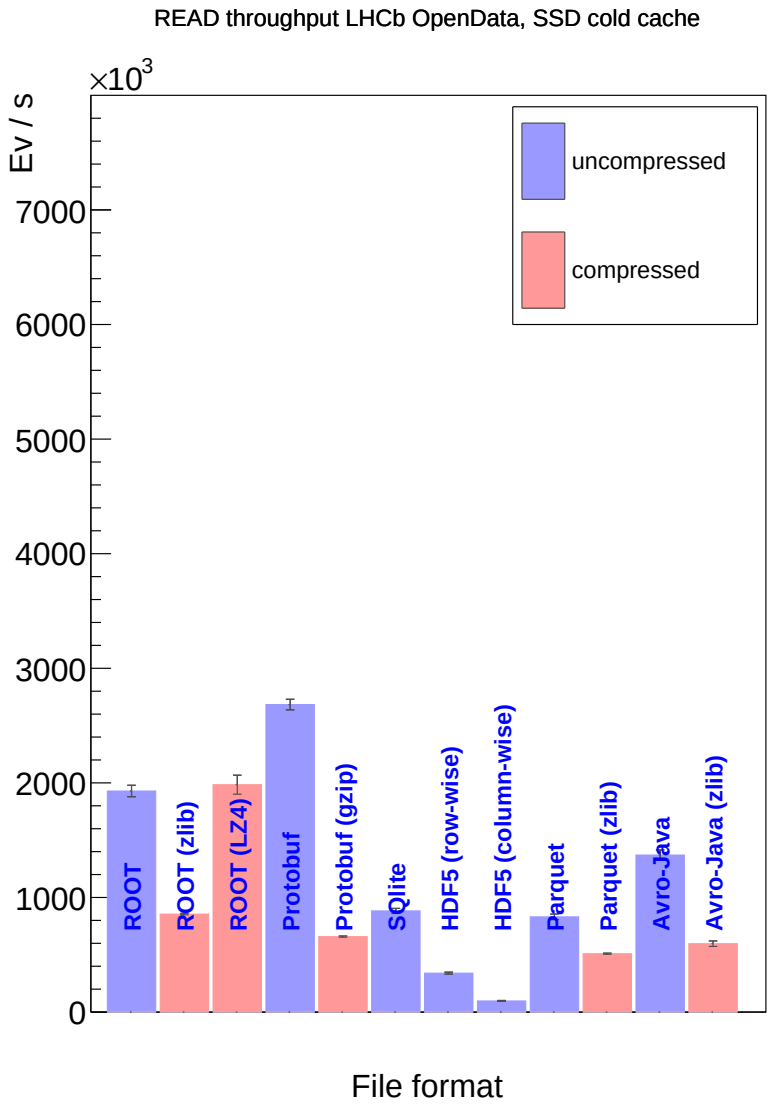


Throughput Plotting 2 Variables (SSD)

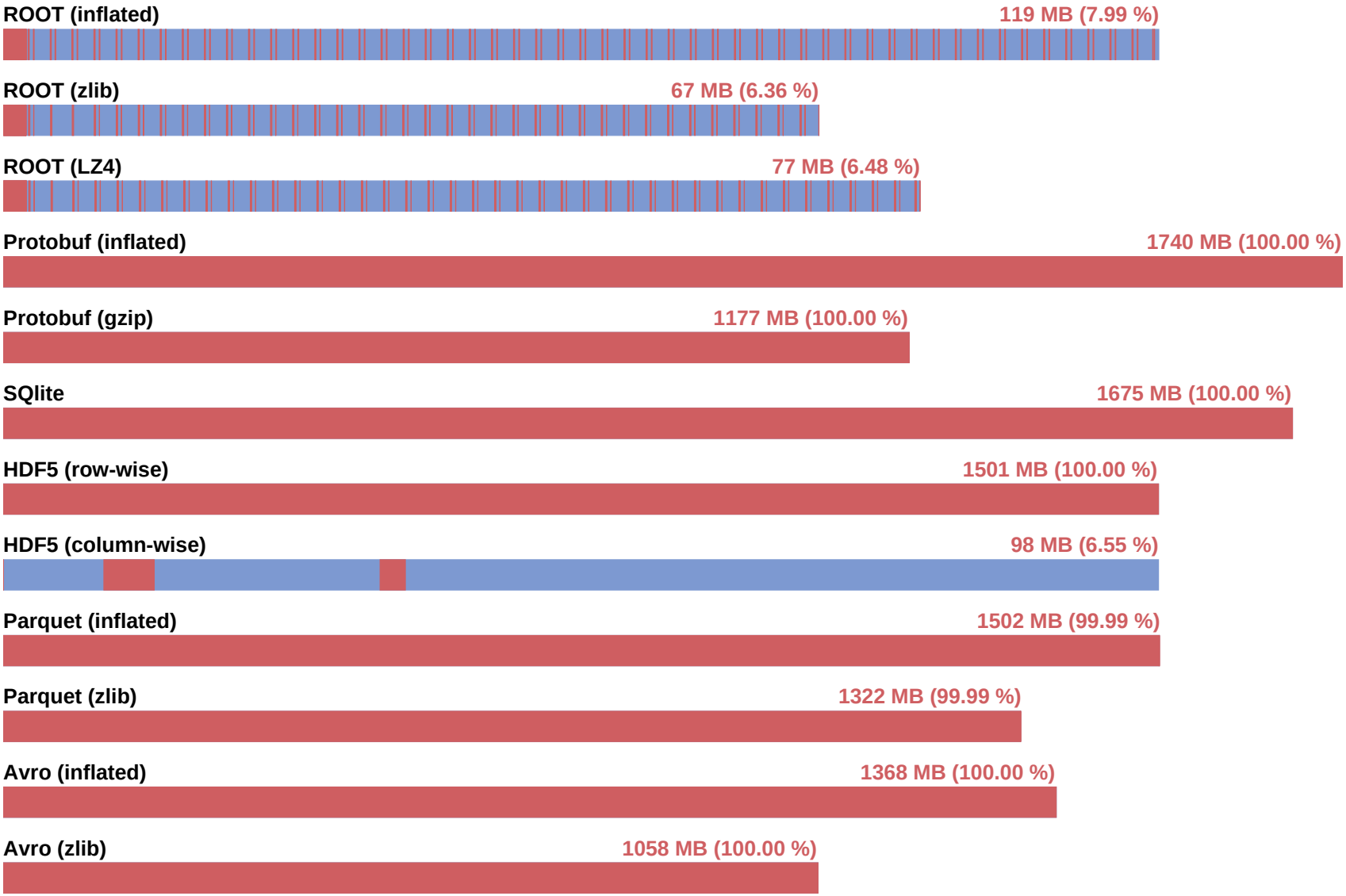


Comparison Reading vs. Plotting (SSD)

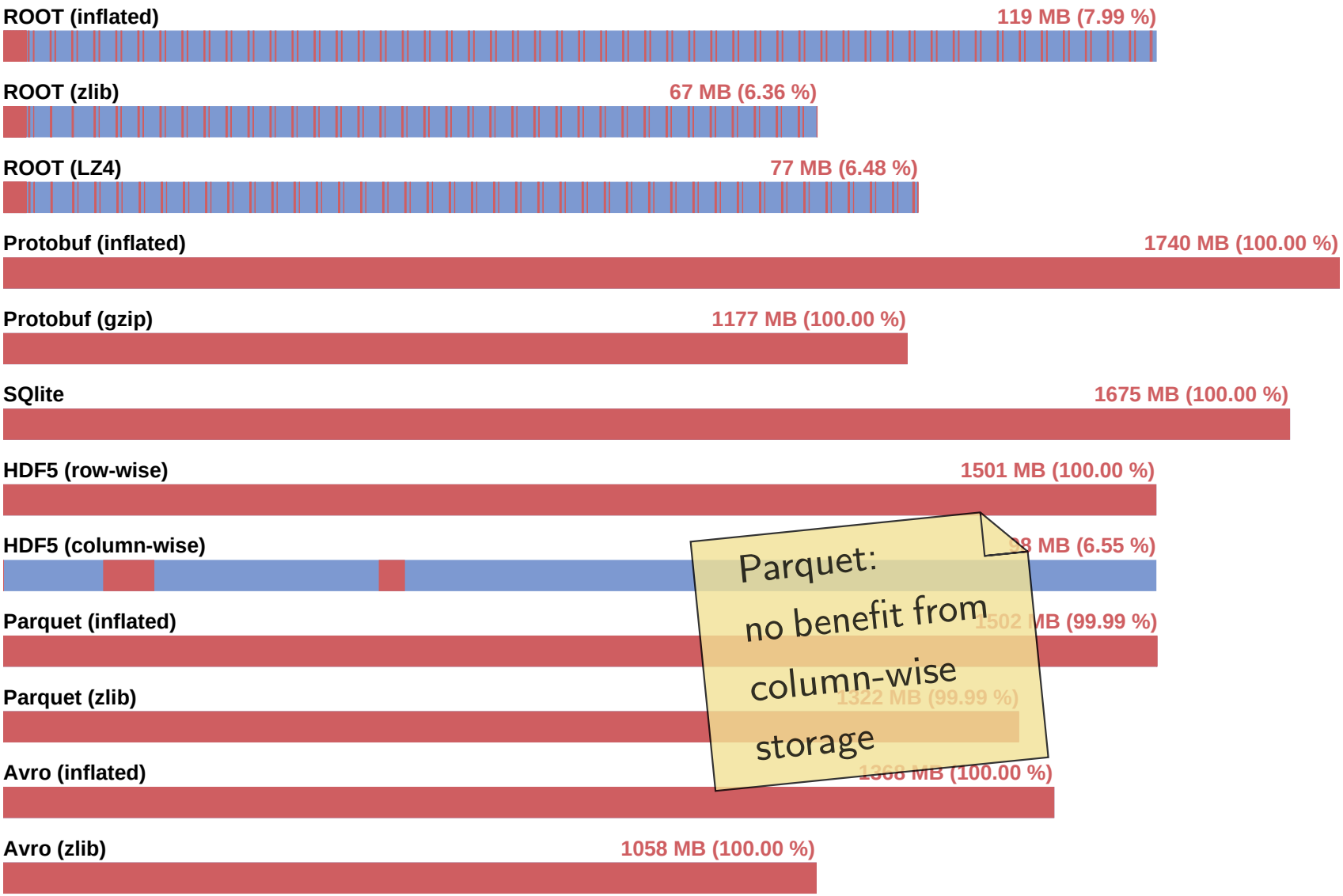




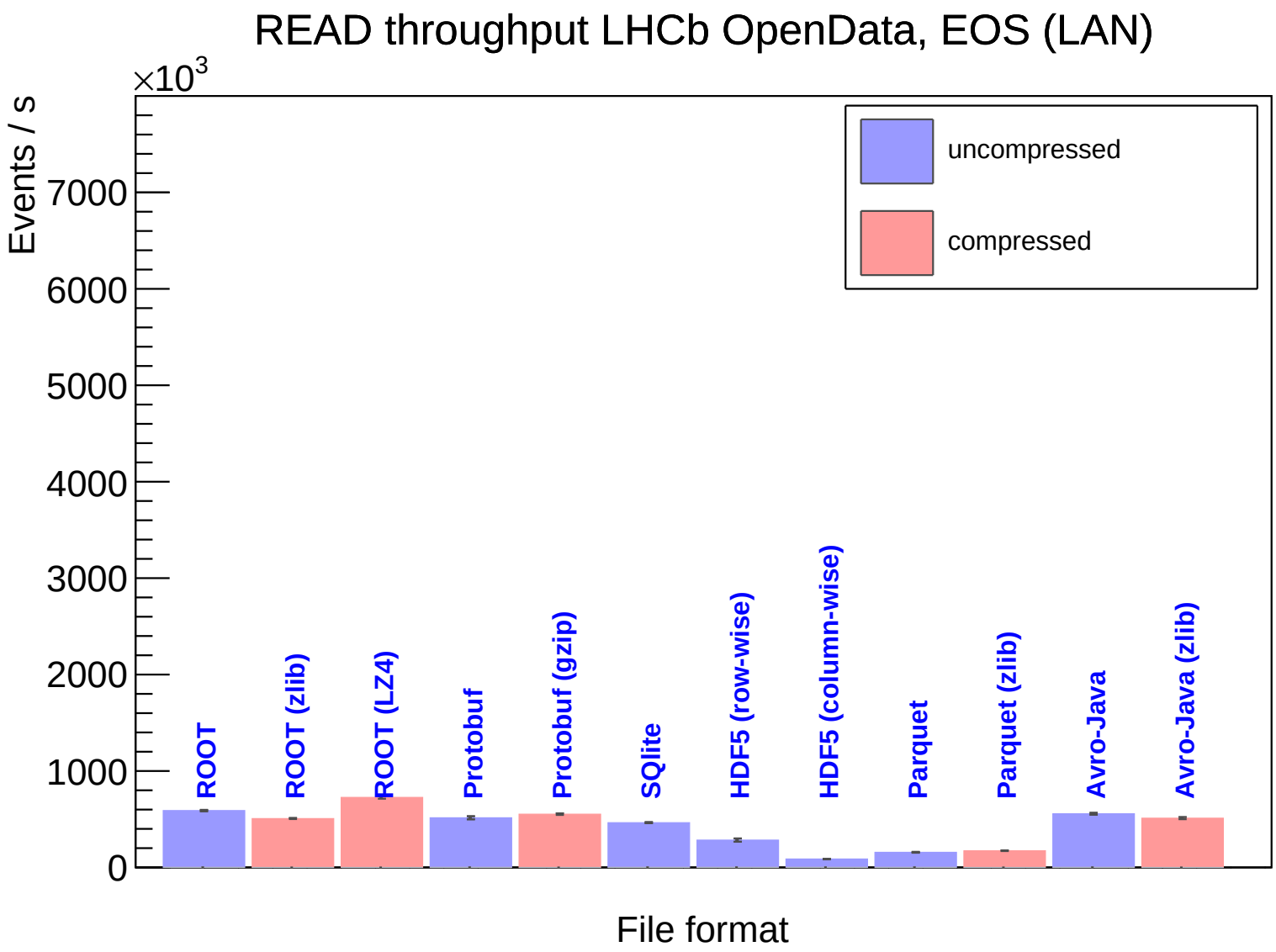
I/O Pattern


ROOT (inflated)
119 MB (7.99 %)
ROOT (zlib)
67 MB (6.36 %)
ROOT (LZ4)
77 MB (6.48 %)
Protobuf (inflated)
1740 MB (100.00 %)
Protobuf (gzip)
1177 MB (100.00 %)
SQlite
1675 MB (100.00 %)
HDF5 (row-wise)
1501 MB (100.00 %)
HDF5 (column-wise)
98 MB (6.55 %)
Parquet (inflated)
1502 MB (99.99 %)
Parquet (zlib)
1322 MB (99.99 %)
Avro (inflated)
1368 MB (100.00 %)
Avro (zlib)
1058 MB (100.00 %)

I/O Pattern

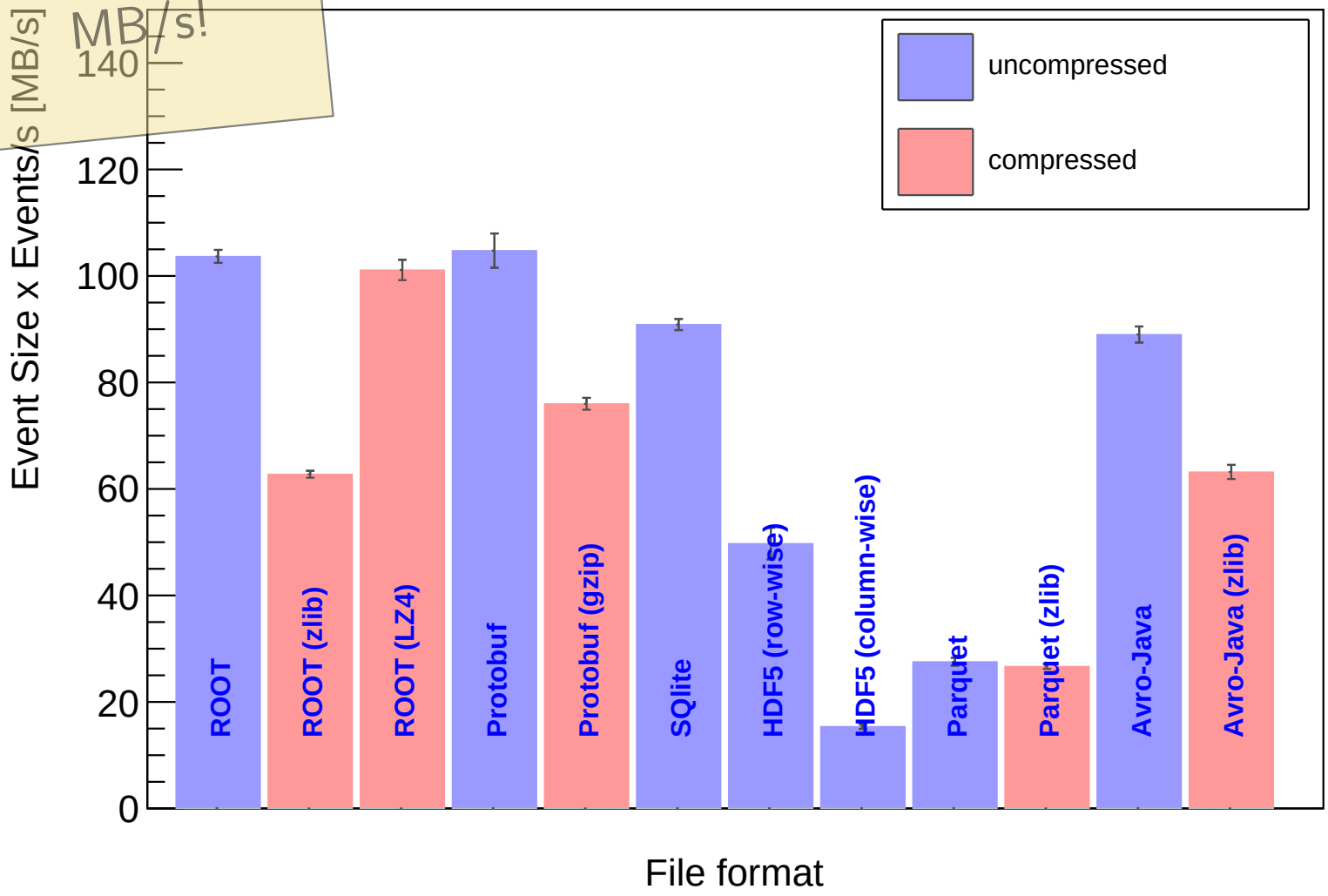

ROOT (inflated)
119 MB (7.99 %)
ROOT (zlib)
67 MB (6.36 %)
ROOT (LZ4)
77 MB (6.48 %)
Protobuf (inflated)
1740 MB (100.00 %)
Protobuf (gzip)
1177 MB (100.00 %)
SQlite
1675 MB (100.00 %)
HDF5 (row-wise)
1501 MB (100.00 %)
HDF5 (column-wise)
Parquet (inflated)
Parquet (zlib)
Avro (inflated)
Avro (zlib)
1058 MB (100.00 %)
Parquet:
no benefit from
column-wise
storage

Read Throughput (EOS, Events/s)



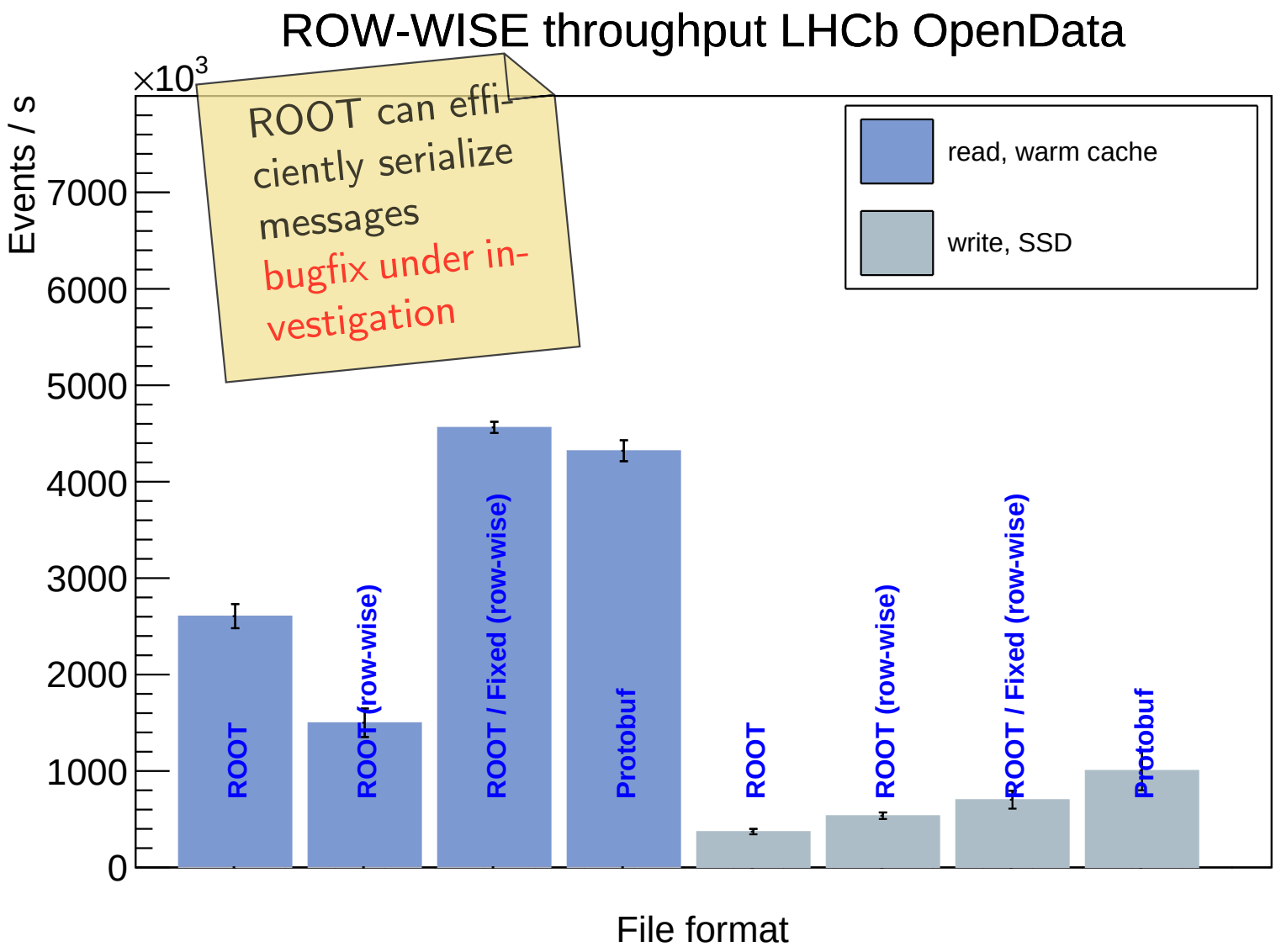
Read Throughput (EOS), Zoomed MB/s

Zoomed, MB/s!

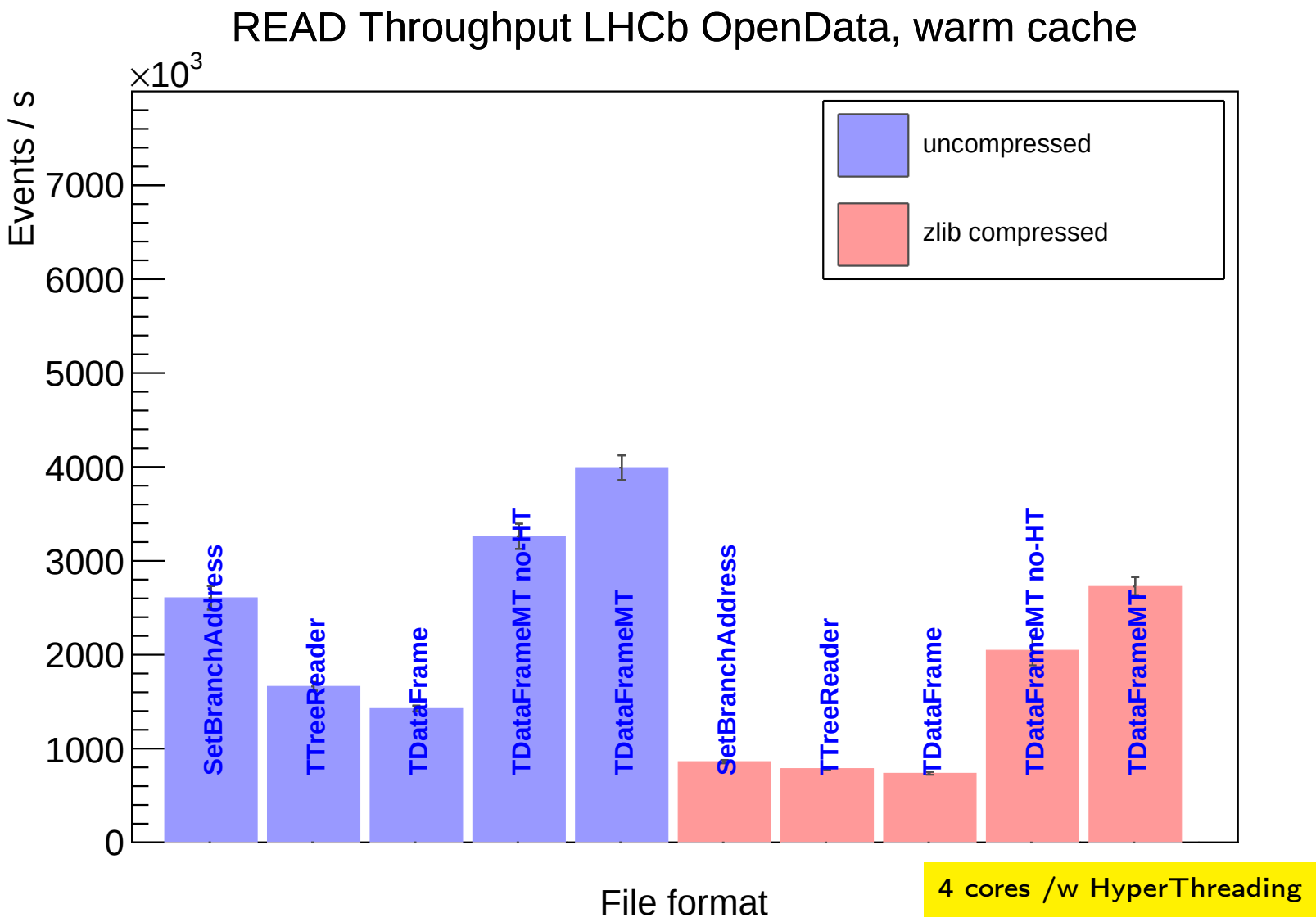


Looking into ROOT I/O

The following results are mostly relevant
on very fast storage or memory

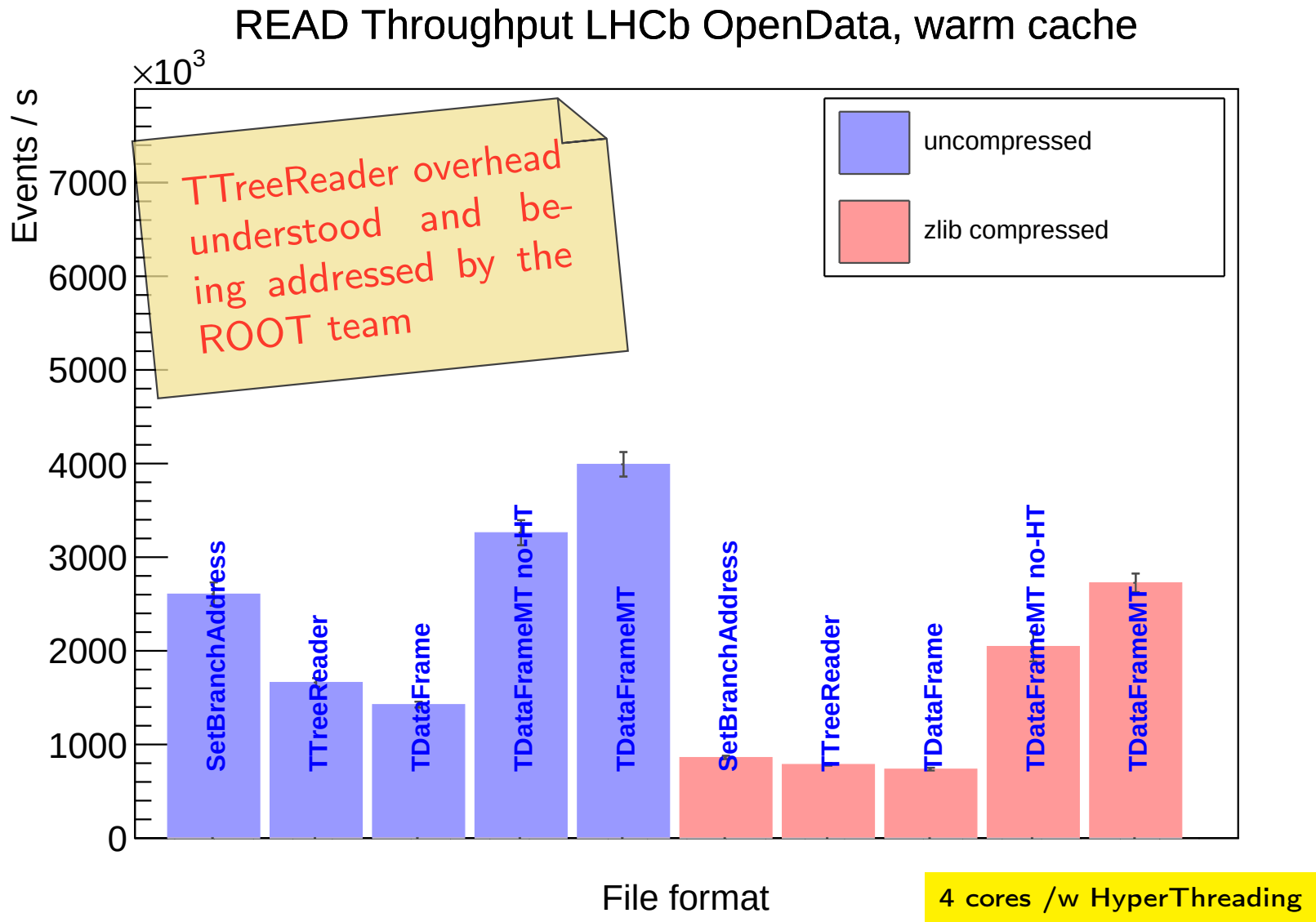

ROW-WISE throughput LHCb OpenData
ROOT can efficiently serialize messages
bugfix under investigation
read, warm cache
write, SSD
Events / s
×10^3
7000
6000
5000
4000
3000
2000
1000
0
ROOT
ROOT (row-wise)
ROOT / Fixed (row-wise)
Protobuf
ROOT
ROOT (row-wise)
ROOT / Fixed (row-wise)
Protobuf
File format

Cost & Benefit of Event Loop Abstractions



TDataFrame: see talk by G. Amadio in track 2

Cost & Benefit of Event Loop Abstractions



4 cores /w HyperThreading

TDataFrame: see talk by G. Amadio in track 2

Cost of Event Complexity

"Flat" Event Structure

```
struct FlatEvent {
  double h1_px;
  double h2_px;
  double h3_px;
  double h1_py;
  ...
};
```

"Deep" Event Structure

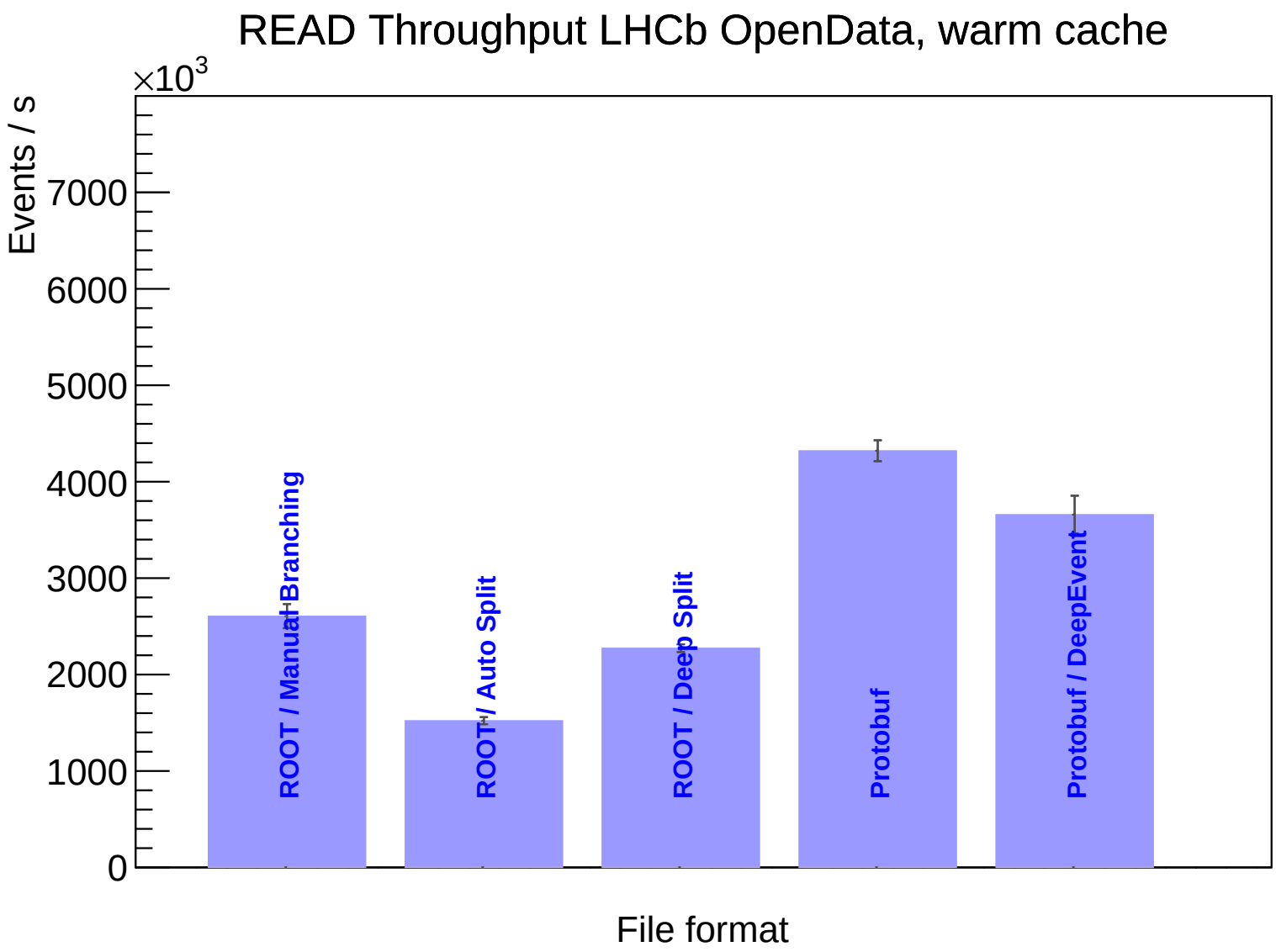
```
struct Kaon {
  double h_px;
  double h_py;
  ...
};
struct DeepEvent {
  std::vector<Kaon> kaons;
};
```

Three options considered

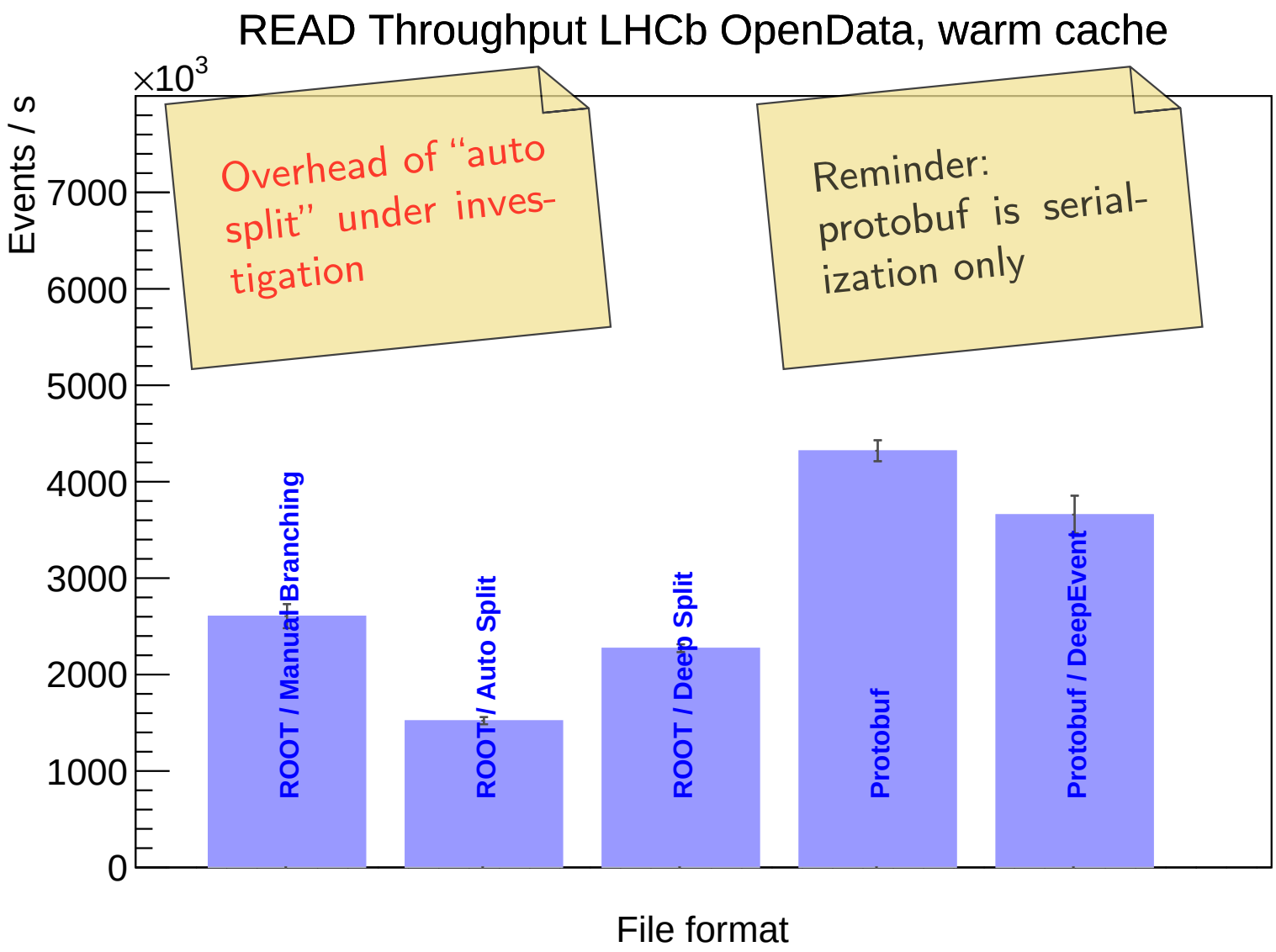
1. "Manual": `FlatEvent`, branches created by hand
2. "Auto": `FlatEvent`, branches created by ROOT with split level 99
3. "Deep": `DeepEvent`, branches created by ROOT with split level 99

(For many branches, manual splitting becomes infeasible)

Cost of Event Complexity / Throughput

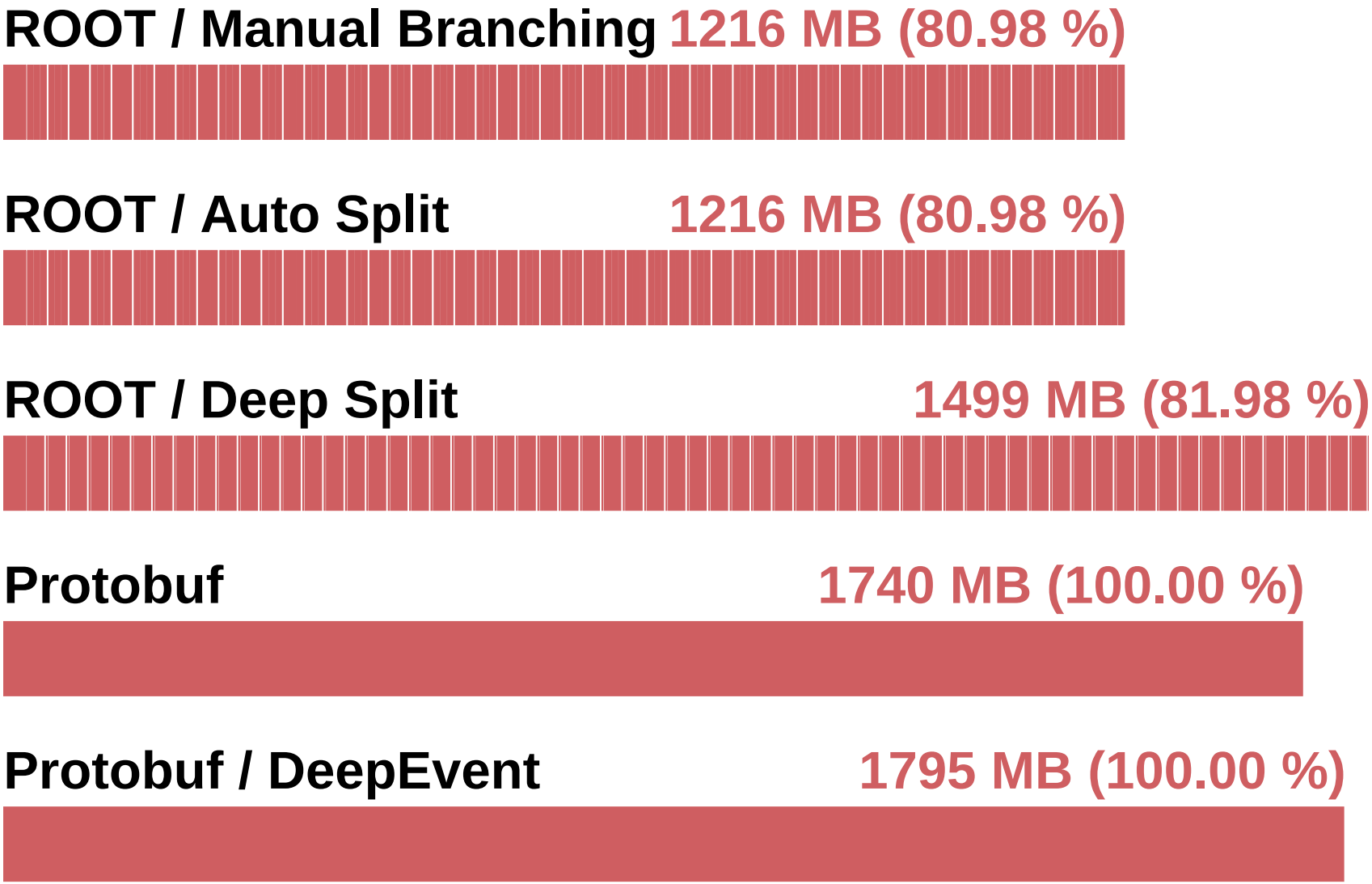


Cost of Event Complexity / Throughput



Cost of Event Complexity / IO Pattern

Reminder: reading 21 of 26 columns



Conclusion

Conclusion

Data Formats

- Large performance differences by different data formats and libraries
- Protobuf is a good benchmark for serialization performance
- HDF5 is most suitable for HPC-style processing of multi-dimensional arrays
- Parquet is an interesting format but C++ libraries are not yet ready

ROOT I/O

- In general best performance in these tests
- Several issues found & being addressed
- LZ4 turns out to be a good trade-off for analysis data sets
- Optionally checksummed ROOT files might be desirable

Many thanks to LHCb for the OpenData data set!

Backup

Benchmark Details

Hardware	Type
CPU	i7-6820HQ @ 2.7 GHz (4 cores + HT)
Memory	2×16 GB DDR4 2133 MHz
SSD (flash)	1 TB Toshiba XG3 PCIe
HDD (spinning)	Western Digital WD20NMVW

Library	Version
ROOT	6.10/04
protobuf	3.3.2
sqlite	3.19.3
hdf5	1.10.0_patch1
avro-c	1.8.2
avro-java	1.8.2
parquet-cpp	1.2.0

Operating system: Linux 4.12, glibc 2.25, gcc 7.1.1, EOS 4.1.26

Storage Technology Variety

Universally efficient data layout and encoding is challenging
given large spectrum of storage performance characterisics

- Latency:
  10 ns RAM → 100 µs 3D XPoint → 1 ms SSD → 10 ms HDD, LAN
  between fastest and slowest: ×100 000
- Throughput:
  20 GB/s RAM → 2 GB/s SSD, 3D XPoint → 100 MB/s HDD, LAN
  between fastest and slowest: ×200

Additionally: uneven scaling of components, e. g.
memory size vs. network bandwidth in laptops

Links

Links

https://github.com/lhcb/opendata-project
https://github.com/lhcb/opendata-project/blob/master/Background-Information-Notebooks/EventData.ipynb
https://github.com/jblomer/iotools/tree/acat17