A quantitative review of data formats for HEP analyses

J Blomer

CERN, Geneva, Switzerland

E-mail: `jblomer@cern.ch`

Abstract. The analysis of High Energy Physics (HEP) data sets often takes place outside the realm of experiment frameworks and central computing workflows, using carefully selected "n-tuples" or Analysis Object Data (AOD) as a data source. Such n-tuples or AODs may comprise data from tens of millions of events and grow to hundred gigabytes or a few terabytes in size. They are typically small enough to be processed by an institute's cluster or even by a single workstation. N-tuples and AODs are often stored in the ROOT file format, in an array of serialized C++ objects in columnar storage layout. In recent years, several new data formats emerged from the data analytics industry. We provide a quantitative comparison of ROOT and other popular data formats, such as Apache Parquet, Apache Avro, Google Protobuf, and HDF5. We compare speed, read patterns, and usage aspects for the use case of a typical LHC end-user n-tuple analysis. The performance characteristics of the relatively simple n-tuple data layout also provides a basis for understanding performance of more complex and nested data layouts. From the benchmarks, we derive performance tuning suggestions both for the use of the data formats and for the ROOT (de-)serialization code.

1. Introduction

The analysis of high energy physics data sets is typically an iterative and explorative process. From centrally curated experiment data and software frameworks, derived data sets are created for particular physics working groups or analyses. These data sets are often stored in the form of ROOT "n-tuples" [1], i. e., tabular data, or modestly nested tabular data (such as a vectors of jets for every event).

In this contribution, we presume that these "final data sets" are small enough to be processed by a single workstation. We further presume that such analysis data sets are independent from the software framework used to create them, so that they can potentially be processed using a variety of tools and data formats.

We evaluate several popular such libraries and data formats from industry and academia. We compare I/O performance, usability, and reliability against storage media failures. We are looking at the use case of I/O dominated, repeated, partial reading of the data set. It should be noted that for reading only a subset of the available columns of the data set, columnar data formats that store columns consequtively (instead of rows) have a natural advantage. As a benchmark, we use a typical LHC Run 1 analysis as described by the LHCb OpenData sample [2].

2. Data Formats and Libraries

The following list provides an overview of the data formats and access libraries that are used in this comparison.

ROOT ROOT [3] stores data in "trees", a collection of serialized, possibly nested C++ objects in columnar layout.

Protobuf Google protobuf [4] is not a data format per se but rather a serialization library for records of simple data types. A simple data format can be constructed, however, by concatenating serialized blobs with a header indicating each record's size.

SQLite SQLite [5] stores data as relational tables and provides data access routines through the SQL language.

HDF5 HDF5 [6] stores multi-dimensional arrays of simple data types. By changing the arrangement of the arrays, the user can effectively store data either column-wise or row-wise. HDF5 is popular in the High Performance Computing community, where it is particularly beneficial through its tight integration with the MPI I/O protocol.

Avro Apache Avro [7] provides row-wise serialization. It is mainly used in the Hadoop ecosystem. In contrast to Google Protobuf, Avro provides a full data format for collections of records out of the box. Its primary access library is written in Java, although a C port exists, too.

Parquet Apache Parquet [8] is a column-wise serialization format mainly used in the Hadoop ecosystem. Access libraries exist for Java and C++.

3. Sample Analysis

The LHCb OpenData sample comprises 8.5 million LHC Run 1 events describing $B^{\pm} \to K^{\pm}K^{+}K^{-}$ decays. Depending on the file format, the data set size is 1.1 GB to 1.5 GB. For every event, 26 values ("branches" in ROOT terminology) of floating point or integer type are stored. In order to approximate the B mass from the recorded Kaon candidates, 21 out of the 26 provided values are required. Furthermore, 2.4 million events can be skipped because one of the Kaon candidates is flagged as a Muon (we apply a cut). As an emulation of *reading the data set*, we calculate the sum of all 21 values of the 6.1 million non-cut events. As an emulation of *plotting from the data set*, we calculate the sum of only 2 values of all events.

Compared to ATLAS xAODs [9] or CMS MiniAODs [10], this is a simple data schema. It helps us understanding the performance base case.

4. Evaluation

With each library and in each format, the data set can be written and read with not more than a few hundred lines of code. No differences occorred in the floating point results. ROOT provides the most seamless integration by directly storing C++ classes. For the other libraries the data schema has to be explicitly specified. Only ROOT and SQLite provide an obvious method for schema evolution, i. e., adding a column to an already existing file. SQLite, on the other hand, provides no support for data compression. While HDF5 does in principal support data compression, it is omitted in these tests because compression requires significantly more code by the user who has to manually break-up the data tables in blocks.

4.1. Resilience against silent data corruption

Data on unmanaged storage is subject to a small, but non-neglible chance of silent data corruption. Silent data corruption is caused by media failures, which remain undetected during normal operation. It can also occur due to transmission errors when copying the data. In order to test the data formats' ability to detect silent corruption, we artificially introduce random bit

Table 1. Number of outcomes of 100 read trials with random bit flips. "No effect" means that the program run produced the correct result. "Crash" means an abnormal termination of the program run. "Err. Msg" means that the word "error" was emitted to stdout, while the program continued. The remaining cases are labeled as "undetected", that is an ordninary program run producing an incorrect result. The undetected errors for ROOT/LZ4 were meanwhile addressed by the ROOT developers.

Format	No effect	Crash	Err. Msg.	Undetected
ROOT (uncompressed)	68	-	-	32
ROOT (zlib)	27	41	32	-
ROOT (lz4)	60	2	4	34
ROOT (lzma)	27	37	36	-
Protobuf (uncompressed)	58	8	-	34
Protobuf (gzip)	-	100	-	-
SQlite	56	12	-	32
HDF5 (row-wise)	63	-	-	37
HDF5 (column-wise)	61	-	-	39
Parquet (uncompressed)	63	4	-	33
Parquet (zlib)	28	72	-	-
Avro-Java (uncompressed)	63	-	-	37
Avro-Java (zlib)	51	27	-	22

flips in the following way. For a reduced data set consisting of the first 500 000 events, we read the corresponding data file one hundred times. For each of these 100 reads, we flip one randomly selected bit in the file beforehand. So effectively we read 100 slightly different files for every data format. Possible results of reading such damaged files range from no change at all, e. g., when the bit flip happens to be at a position that is skipped during reading, to a crash of the program. A malicous result is a difference in the physics result, where no indication of a failure is given during the reading.

Table 1 summarizes outcome of the tests. For all of the data formats, except Avro, bit flip protection is a side effect of the checksums of the compression algorithm. Without compression, all data formats are subject to an occasional, undetected change of the physics result in case of silent corruption. Avro uses the zlib "raw mode" which explicitly omits the checksum even for compressed data. In ROOT, some of the bit flips trigger only error messages but no crash, which might be overlooked by a not carefully programmed application.

4.2. Performance

All performance measurements are performed on a machine with a 4 core Intel i7-6820HQ with hyper-threading, 2×16 GB RAM, a 1 TB Toshiba XG3 PCIe SSD and Gigabit Ethernet, which is supposed to resemble the performance of a typical workstation. The system runs a Linux 4.12 kernel, glibc 2.25, gcc 7.1.1 and the EOS 4.1.2 client [11]. The source code of the benchmarks is available on github[1].

Figure 1 shows the encoding effeciency of the different file formats. For this data set, the efficiency of compression is of the order of 20 % to 35 % while the difference between compression algorithms is of the order of 10 % to 15 %. Small file sizes are particularly performance-relevant for slow storage devices such as spinning hard drives or Gigabit attached storage.

[1] `https://github.com/jblomer/iotools/tree/acat17`

Figure 2 shows differences by a factor 10 and more between the file formats in reading throughput from fast, memory-class storage. For Avro, the Java library is used instead of the C library due to a tenfold performance penalty that is observed by the current C implementation. On fast storage, uncompressing data dominates the read spead, with the notable exception of the LZ4 compression algorithm.

The HDF5 column-wise performance penalty can be explained by a large number of seek calls due to the use of unchunked tables (cf. Figure 5). This is in contrast to the ROOT and Parquet columnar layouts, where the columnar layout is not applied globally but on blocks of a certain number of consequitve rows. These row groups are read into memory in a single go to avoid seeking on the storage medium.

Figures 3 and Figure 4 show the effect of a very sparse reading of columns. As expected, there is a performance boost for the columnar data formats while the performance of the row-wise formats remains more or less constant. The performance improvement of Parquet, although columnar, is surprisingly small. An analysis of the read pattern in Figure 5 reveals that the entire Parquet file is being read even though only a subset of columns is requested. This behavior is not an inherent problem of the data format but it can be traced back to the use of memory mapped I/O in the Parquet library.

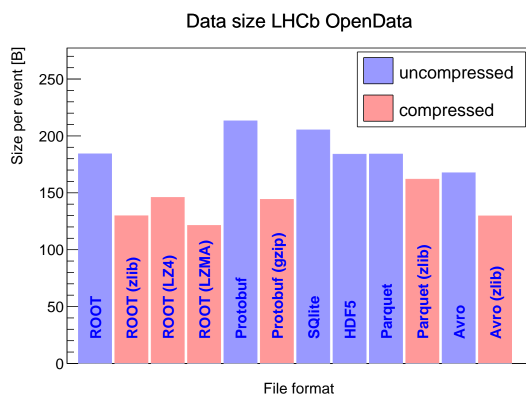


Figure 1. Encoding efficiency of the file formats.

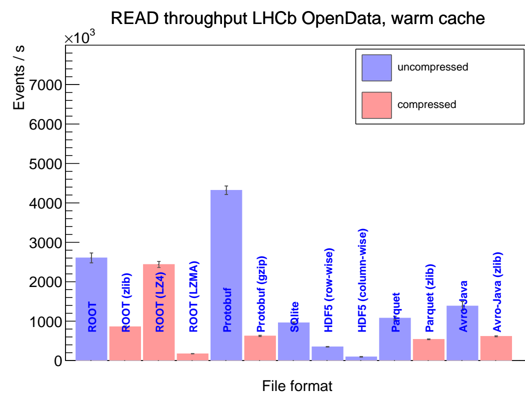


Figure 2. Throughput of reading data from from fast, memory-class storage.

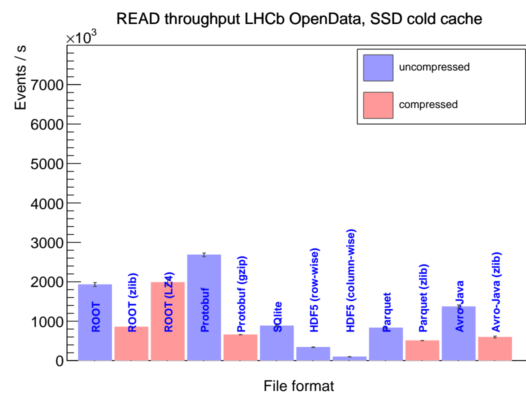


Figure 3. Throughput of reading full event data from the SSD with a cold hard disk cache.

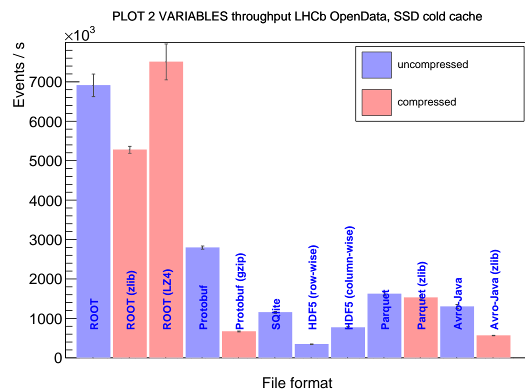


Figure 4. Throughput of sparsely reading data, typically done for plotting data.

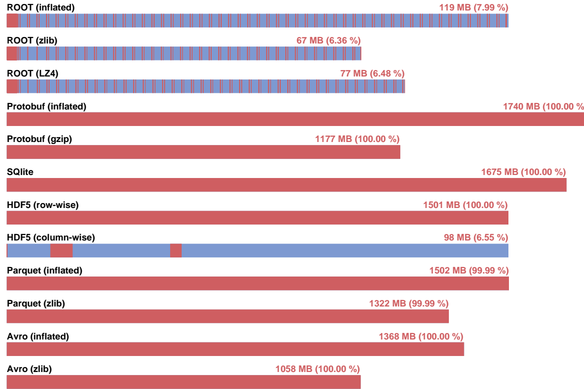


Figure 5. Visualization of read parts (red blocks) and unread parts (blue blocks) of the data file for the benchmark of plotting two columns. Data is collected by a recording fuse [12] file system.

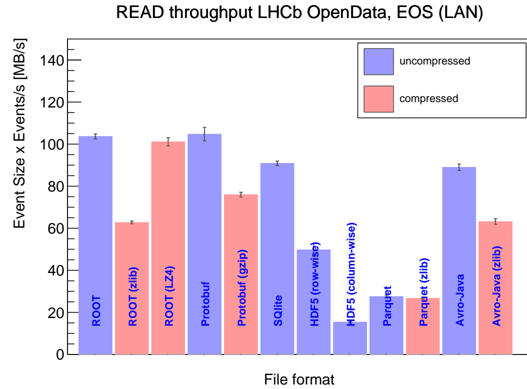


Figure 6. Throughput of reading data from an EOS mountpoint through a 1 GbE link with 20 ms round-trip time. Note that the throughput is shown in MB/s pointing out the network interface as a bottelneck.

Figure 6 shows that for the fastest file formats, ROOT and Google Protobuf, the 1 GbE network interface card becomes a bottleneck. Further tests with network latency increased by traffic shaping show that for high-latency (and thus low-throughput) links, the number of read bytes quickly becomes the dominating factor for performance.

5. Comparison of ROOT file format options and access modes

In this section, we look at the performance impact of different ways of using ROOT. All the following tests are performed on data in warm file system buffers in order to resemble fast storage devices and to emphasize performance differences.

Figure 7 shows the impact of column management to the ROOT serialization speed. When it is known beforehand that all of the columns of a data set need to be read, ROOT can be instructed to drop the columnar storage layout by setting the data record's "split level" to zero. In this test, ROOT has a significant overhead due to handling of potentially self-referencing data records. The overhead can be avoided if, for instance, the data record contains only simple, non-pointer data members. As indicated by the "fixed" data point, in this case ROOT has serialization performance comparable to Protobuf. Note that a proper patch has not yet been sent to the ROOT project though.

Figure 8 shows the impact of different event loop abstractions. Data values can be directly copied into memory areas ("SetBranchAddress") or, through the TTreeReader interface, bound to C++ variables in a type-safe manner. An unnecessary overhead in the type-safety checking of TTreeReader has been identified by the ROOT team and is being followed up. The TDataFrame abstraction, built on top of the TTreeReader interface, provides an interface similar to Python pandas. In this test, it shows some threefold speed-up on 4 cores for its automatic concurrent iteration through the data set. This is particularly beneficial to speed-up decompression.

Figure 9 shows the impact of ROOT splitting options and data record complexity. Two possible C++ representations of the LHCb OpenData data set, FlatEvent and DeepEvent, are sketched one the right hand side of the figure. We presume that particularly for large data records, users make use of ROOT's capability to automatically create columns from C++ class members. This "splitting" does not significantly change that storage layout compared to manual creation of branches. Some of the performance overhead of the DeepEvent class is due to larger

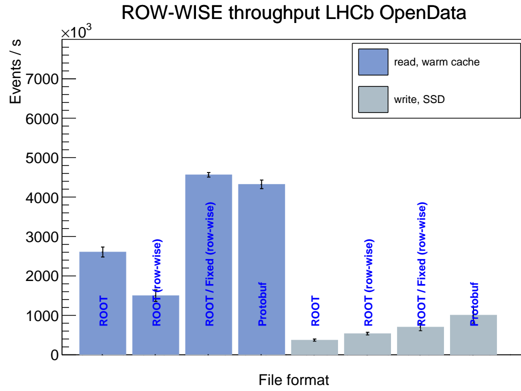


Figure 7. Read and write throughput of pure serialization of data with split level 0 in ROOT compared to Protobuf. "Fixed" refers to a patch avoiding additional code paths related to data records with pointers.

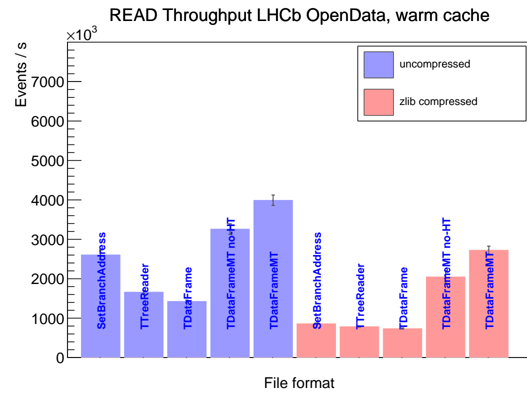
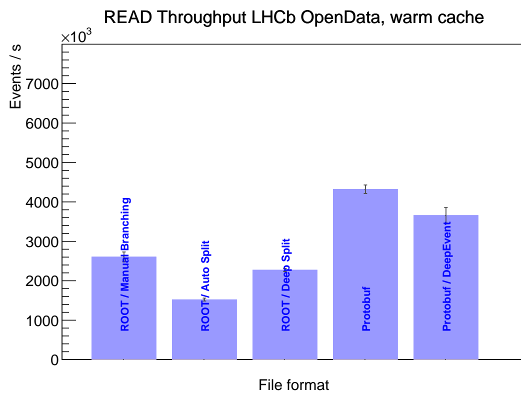


Figure 8. Comparison of event loop abstractions. The "MT" suffix indicates multi-threaded TDataFrame use, the "no-HT" suffix indicates that hyper-threading is turned off.




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

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

Figure 9. Impact of ROOT splitting options and event complexity. "Manual branching" refers to explicit creation of columns corresponding to the members of FlatEvent. "Auto Split" refers to FlatEvent data records that is parsed and atomatically transformed into columns by ROOT. "Deep Split" refers to DeepEvent data records automatically serialized by ROOT.

file sizes, as every data record also needs to store the size of the Kaon vector. This is reflected in both ROOT and Protobuf. The larger overhead of splitting the simpler data records of type FlatEvent compared to the DeepEvent data records is still under investigation.

6. Conclusion

The benchmarks in this paper show that the turn-around time of physics analyses can be very sensitive to the data format and access library. Overall, ROOT outperforms all other libraries except for few corner cases. In particular, pure serialization is faster with Protobuf, which benefits from not having to deal with column management.

Several smaller issues in ROOT's I/O code were identified in the course of these benchmarks and are currently followed up. The new TDataFrame abstraction for event loops is not only interesting for allowing for more more concise user code but also for its ability to automatically parallelize event loops. Currently, this is a unique ROOT feature. The new LZ4 compression

algorithm provides a very interesting trade-off between storage efficiency and processing speed. In particular, for the evolving close-to memory-class storage such as 3D X-Point, LZ4 occurs as the best choice for analysis data sets.

7. Acknowledgements

I want to especially thank Philippe Canal and Axel Naumann for all their help in analyzing ROOT's behavior. I want to thank Jim Pivarski for pointing out the Avro Java library to me and for following up on Parquet's memory mapped I/O behavior with the developers. I want to thank Hervé Rousseau from CERN IT for providing access to their EOS test instance.

References

[1] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A* **389** 81–86

[2] Rogozhnikov A, Ustyuzhanin A, Parkes C, Derkach D, Litwinski M, Gersabeck M, Amerio S, Dallmeier-Tiessen S, Head T and Gilliver G 2016 Particle physics analysis with the cern open data portal talk at the 22nd Int. Conf. on Computing in High Energy Physics (CHEP'16)

[3] Rademakers F, Brun R *et al.* 2017 ROOT - An Object-Oriented Data Analysis Framework. root-project/root: v6.10/04 URL `https://doi.org/10.5281/zenodo.848819`

[4] The Google Protobuf Project, 2017 "protobuf" [software], version 3.3.2 URL `https://github.com/google/protobuf/tree/v3.3.2`

[5] The SQLite project, 2017 "sqlite" [software], version 3.19.3 URL `https://www.sqlite.org/src/info/0ee482a1e0eae22e`

[6] The HDF Group, 2017 "hdf5" [software], version 1.10.0 patch 1 URL `https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.10/hdf5-1.10.0-patch1/`

[7] The Apache Avro project, 2017 "avro" [software], version 1.8.2 URL `https://avro.apache.org/releases.html`

[8] The Apache Parquet project, 2017 "parquet-cpp" [software], version 1.2.0 URL `https://github.com/apache/parquet-cpp/tree/apache-parquet-cpp-1.2.0`

[9] Buckley A, Eifert T, Elsing M, Gillberg D, Koeneke K, Krasznahorkay A, Moyse E, Nowak M, Snyder S and van Gemmeren P 2015 *Journal of Physics: Conference Series* **664**

[10] Petrucciani G, Rizzi A and Vuosalo C 2015 *Journal of Physics: Conference Series* **664**

[11] The EOS project, 2017 "eos" [software], version 4.1.26 URL `https://github.com/cern-eos/eos`

[12] Henk C and Szeredi M Filesystem in Userspace (FUSE) `https://github.com/libfuse/libfuse` URL `https://github.com/libfuse/libfuse`