Columnar data analysis with ATLAS analysis formats

Nikolai Hartmann[1,*], *Johannes* Elmsheuser[2], and *Günter* Duckeck[1]on behalf of ATLAS Software and Computing

[1]Ludwig-Maximilians-Universität, München, Germany
[2]Brookhaven National Laboratory, Upton, NY, USA

Abstract. Future analysis of ATLAS data will involve new small-sized analysis formats to cope with the increased storage needs. The smallest of these, named DAOD_PHYSLITE, has calibrations already applied to allow fast downstream analysis and avoid the need for further analysis-specific intermediate formats. This allows for application of the "columnar analysis" paradigm where operations are applied on a per-array instead of a per-event basis. We will present methods to read the data into memory, using Uproot, and also discuss I/O aspects of columnar data and alternatives to the ROOT data format. Furthermore, we will show a representation of the event data model using the Awkward Array package and present proof of concept for a simple analysis application.

1 Introduction

The increasing amounts of data from the ATLAS [1] experiment in the coming LHC Run 3 and even more so in the future High-Luminosity LHC (HL-LHC) data taking era [2] demand for efficient processing to be able to analyze these data with the available computing resources. Finding a compromise between productivity of analyzers and highly optimized data analysis code is important. Columnar data analysis allows for such a compromise by separating the low-level numerical calculations on large arrays from the analysis code (typically written in python) which describes operations that process an array at a time. In contrast, the more traditional approach in high energy physics (HEP) consists of manually writing event loops that necessitate the usage of compiled programming languages like C++ to produce efficient code. This code is often time consuming to develop.

With the increasing demand for processing large amounts of data in fields outside of HEP, and the rising popularity of machine learning methods, the availability of general tools, especially in the scientific python ecosystem, has increased. Additions to this ecosystem, inspired by HEP needs, have made significant progress in recent years. Many of them are collected under the umbrella of the Scikit-HEP [3] project. Currently, the Coffea framework [4], together with the Awkward Array package [5], provide the most complete set of tools for columnar data analysis in HEP.

*e-mail: nihartma@cern.ch

© 2021 CERN for the benefit of the ATLAS Collaboration. CC-BY-4.0 license.

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

The ATLAS analysis model [6] for the LHC Run 3 includes new small-sized analysis formats DAOD_PHYS and DAOD_PHYSLITE. The latter has a set of calibrations and reconstructions, harmonized across common analysis use cases, already precalculated which allows running fast analysis without the need to call complex reconstruction and calibration algorithms. DAOD_PHYSLITE is therefore an ideal input dataset for columnar data analysis, as discussed in this document.

2 I/O and Storage formats

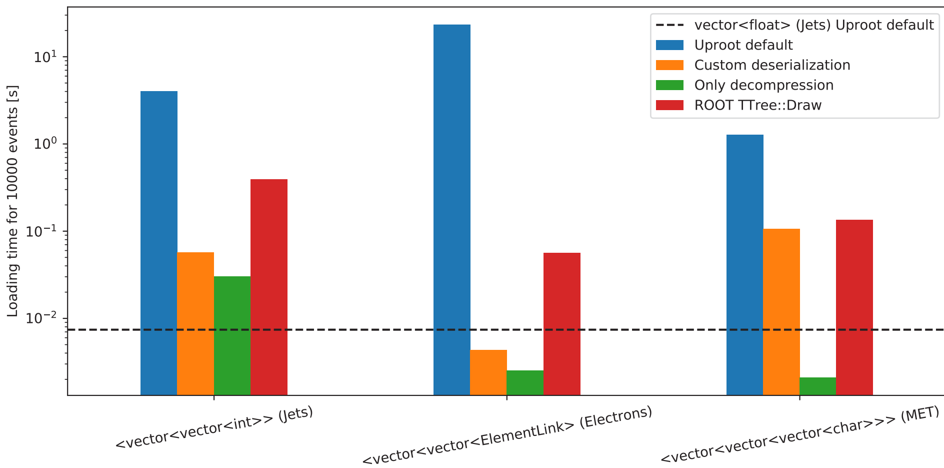

Figure 1. Loading times[1] for nested structures where data content and offsets (list lengths) cannot be read separately from stored columns. The times include the time needed for decompression. The decompression time without deserialization is shown for reference, as well as an example for a branch that can be read in a columnar fashion, indicated by a dashed black line. Using the improved deserialization routines, complex data structures can be read efficiently, with similar or better performance than ROOT's `TTree::Draw` function. All times are recorded after running a second time to load from cached pages instead of hard disk.

Python data science tools mostly deal with flat n-tuples (table-like formats). However, HEP data like DAOD_PHYSLITE contains variable length lists of numbers per event (so-called single-jagged vectors), e.g. list of tracks, leptons etc. The ROOT [7] data format, currently used for DAOD_PHYSLITE, allows for columnar reading of such single-jagged structures, since event offsets are stored separately from the data. Structures with multiple members can also be automatically split into separate columns, which is done for DAOD_PHYSLITE where possible. The Uproot [8] package can therefore read these quantities efficiently using NumPy [9]. More complex are cases with a higher level of nesting, where offsets are not stored separately in the ROOT file format. A typical example for this in DAOD_PHYSLITE are references into other collections - called `ElementLink`. For example, an electron can have multiple tracks associated, thus needing a double-jagged vector (`vector<vector<ElementLink>>`) to represent these indices. Furthermore, `ElementLink` structures that are more than single-jagged are not split into their members (an index and a

[1] Performance timings in this document were measured using a Laptop with SSD disk and an Intel Core i5 CPU with 4 physical cores.

hash representing the collection linked to). Reading such complex data structures requires routines that iterate through the data, collecting the offsets and data. The current python code used for this in Uproot needs to be replaced by compiled routines for efficient reading. Figure 1 shows an example using a python implementation with just-in-time compilation using numba [10]. This confirms the findings shown in Ref. [5]. Notably, the loading time is always lower than reading with ROOT's `TTree::Draw` function. The performance for reading the DAOD_PHYSLITE format with Uproot will therefore not be limited by deserializing branches with higher level nesting when such specialized routines are used. They are foreseen to be included in future versions of the Awkward Array package.

Reading data into a columnar representation in memory also poses the question what the best storage format for this workflow is. With splitting of complex objects into members and grouping their data into contiguous blocks (called baskets), the ROOT format is well suited for columnar data access. However, the basket size needs to be chosen appropriately. Columnar data analysis workflows can benefit from as large as possible blocks of contiguous data. Besides better compression efficiency, this can lead to higher throughput when reading data from disks, especially in case only few of all available columns are read, which is common in ATLAS data analysis. In environments with parallel disk access of multiple processes the reduced amount of disk seeks is a major factor for this aspect. Figure 2 shows a visualization of the data access patterns when reading only few of all available columns, comparing the currently used ROOT settings for DAOD_PHYSLITE with comparably small basket sizes to the Apache Parquet format [11] with one row group (effectively storing each column in one contiguous block) and ROOT with large baskets (one basket per column).

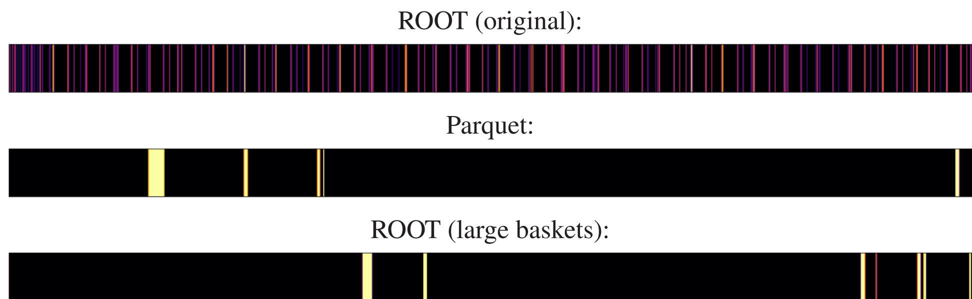

Figure 2. Data access patterns when reading few (in the order of 1 %) of all (approximately 1000) columns. The amount of disk seeks is determined by the basket size in ROOT files and the number of row groups in Parquet files (1 in this example).

When only whole-column access is needed, storage formats other than ROOT (potentially much simpler ones) allow for significantly faster reading (see Table 1). The slower reading of the ROOT format, using Uproot, when comparing to other formats with similar compression settings, can be attributed to the higher complexity of the ROOT format and thus a higher (constant) overhead in addition to the pure data reading. Formats without support for nested/jagged structures (namely HDF5 [12] and zipped NumPy arrays [13]) can be used as storage via Awkward Array's `to_buffers`/`from_buffers` feature to split data into contiguous arrays and reconstitute them via a JSON [14] form. In addition to faster reading, pure columnar storage is easier to compress, which results in smaller file sizes, especially for compression algorithms with high compression ratios such as gzip. Deduplication of offsets (e.g. storing the number of electrons only once instead of for each electron property) can reduce the file size even more (compression alone can't achieve that since each column

Table 1. Size on disk and execution time for reading ≈ 1000 columns of a file with 10000 events into an Awkward Array in memory using different storage formats. The execution times given refer to the minimum out of 5 consecutive trials. The table column "Dedup. offsets" indicates whether offset indices are deduplicated using the `awkward.zip` function (e.g. the number of electrons per event is only stored once instead of for each electron property)

Format	Compression	Dedup. offsets	Size on disk	Execution time
(Up)root	zlib	No	117 MB	6.0 s
(Up)root (large baskets)	zlib	No	116 MB	5.0 s
Parquet	snappy	No	121 MB	0.6 s
Parquet	snappy	Yes	118 MB	0.6 s
HDF5	gzip	No	101 MB	2.0 s
HDF5	gzip	Yes	89 MB	1.6 s
HDF5	lzf	No	137 MB	1.5 s
HDF5	lzf	Yes	113 MB	1.1 s
npz	zip	No	92 MB	2.0 s
npz	zip	Yes	82 MB	1.5 s

is compressed separately). Choosing a format depends on the types of analysis workflows foreseen for this data. A compromise has to be found if both event-by-event and columnar workflows are used on the same data format. Many of the shortcomings in the ROOT file format for pure columnar reading are addressed by the RNtuple Class in ROOT 7 [15] which is another interesting option for the future.

3 Columnar representation of ATLAS analysis data

The ATLAS event data model used in DAOD_PHYSLITE is converted into columnar form using Awkward Array. Many concepts used are similar to the `NanoEvents` module in Coffea, that has been developed for the NANOAOD [16] format at CMS [17]. The top level structure is a `RecordArray` of all available collections (like Electrons, Muons, Jets, etc.). The collections are all of the same length along the first dimension, corresponding to the number of events. This allows for filtering or masking of all collections with event level selections. Jagged arrays are represented by `ListOffsetArray` and cross references by `IndexedArray`. Behavior can be added to collections mimicking e.g. Lorentz vectors. The vector module of the Coffea framework provides classes for Lorentz vectors that can be extended. An example for such an extension is an `xAODTrackParticle` that dynamically calculates the 4-momentum components from the track parameters. Another example is dynamically added cross references, e.g. the associated track particles for electrons. The usage of `VirtualArray` allows for "lazy loading" of columns from the input source only once they are used, allowing for interactive/incremental development. Assuming the data for a certain number of events is represented by an object named `events`, this representation allows for python analysis code such as,

```
>>> import awkward as ak
>>> events[ak.num(events.Electrons) >= 1].Electrons.pt[:, 0]
<Array [7.36e+03, 8.84e+04, ... 3.27e+04] type='20194 * float32'>
```

which is an example for selecting the transverse momentum of the first electron in events with at least one electron. The longitudinal impact parameter of the tracks associated to each electron could be selected with,

```
>>> events.Electrons.trackParticles.z0
<Array [[[-47]], [], ... ] type='50000 * var * var * float32'>
```

which will return a 3-dimensional array (the last 2 dimensions of variable length). Since the `trackParticles` record has the described `xAODTrackParticle` behavior associated, the following will calculate the transverse momenta of the corresponding tracks dynamically from the track parameters.

```
>>> events.Electrons.trackParticles.pt
<Array [[[7.24e+03]], ... 3.09e+04, 4.88e+03]]]
type='50000 * var * var * float32'>
```

4 Proof of concept

To test the columnar analysis workflow for a realistic analysis example, the representation from Section 3 is used to perform a set of object selections for electrons, muons and jets on a Monte Carlo sample of 50000 simulated events of $t\bar{t}$ decays in DAOD_PHYSLITE format. A C++ Athena [18] algorithm using the `SUSYTools` package [19] is used as a reference for comparison. The selection criteria in the columnar analysis are implemented to match the C++ reference as close as possible, but some details like track parameter selections are omitted. This object selection consists of two types of operations. First, filters on stored quantities like isolation variables, momenta and pseudorapidy values have to be applied. The convention from `SUSYTools` defines objects satisfying loose "baseline" and tighter "signal" criteria. Second, overlap removal for ambiguity resolution and additional isolation has to be applied. This is more complex, since it involves checking all combinations of 2 particles. The calculation uses the `awkward.cartesian` function that constructs these combinations. Distance or index based matching can then be performed in a vectorized fashion. As shown in Figure 3, for most selections, the number of objects passing the `SUSYTools` selection can be reproduced by a columnar data analysis in python.

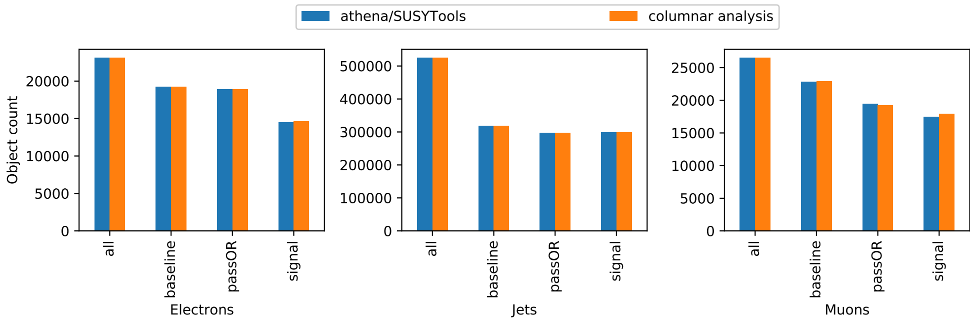

Figure 3. Comparison of object selections on electrons, muons and jets from the `SUSYTools` framework, reproduced by a columnar data analysis using DAOD_PHYSLITE. Further work is required to get full agreement in all selections, most importantly for Muons. Criteria not considered so far in the columnar analysis include filters on track parameters which could explain part of the disagreement.

The processing times for the columnar data analysis are also significantly smaller than the reference, confirming the efficiency of this approach. A comparison is shown in Table 2. A quick test on a different data sample showed that the processing time roughly scales with the

Table 2. Processing time for 50000 DAOD_PHYSLITE events using the reference implementation with Athena/SUSYTools in C++, compared to a columnar data analysis in python. In both cases, only object selections for electrons, muons and jets are evaluated. The total time does not include a constant overhead for initializing the necessary objects and environments before loading (non-meta) data and event processing. This overhead amounts to approximately 15s for Athena/SUSYTools and 2.4s for the columnar data analysis. The measurement "Columnar (cached)" refers to processing with all data already decompressed, deserialized and loaded into memory. This time is significantly lower than the total time that includes loading, indicating that most of the time is spent for input, decompressing and deserialization for the columnar data analysis. All times refer to the minimum of 5 consecutive runs.

Measurement	Total time [s]	average no. events / s
Athena/SUSYTools	22	2300
Columnar	3.8	13000
Columnar (cached)	1.2	42000

mean number of jets. Average data samples are therefore expected to be processed slightly faster than the tested $t\bar{t}$ Monte Carlo sample.

While this first demonstration of an object selection for electrons, muons and jets was successful, many steps of a typical analysis remain to be done. Some of them will require further research and development. One example is the calculation of the missing transverse momentum that is built depending on the analysis specific object selections and has therefore also be addressed in an analysis of the DAOD_PHYSLITE format. From experience with the event-loop-based analysis this is expected to increase the processing time by up to a factor of 2. Another important topic is the treatment of systematic uncertainty variations for later statistical analysis. To be able to calculate these in a columnar data analysis they have to be applicable in a parameterized fashion without the need for running complex reconstruction and calibration algorithms.

5 Summary

Columnar data analysis offers an attractive way to analyze the new ATLAS data analysis format, DAOD_PHYSLITE. Reading data into memory, representing the event data model in columnar format and typical analysis workflows can be done with available python tools. Such a workflow was shown to be efficient, offering significantly faster throughput compared to traditional analysis. However, further research and development is needed to cover all aspects of a full analysis. Due to the efficient processing such an analysis is mainly limited by input reading. Therefore, further tuning of storage settings like ROOT basket sizes could improve the throughput even more, especially when considering parallel file access from many workers. When a format is only used for columnar data analysis, storage formats other than ROOT can be an interesting, performant option. Such alternative formats could also provide substantial savings in terms of storage space, the most critical factor for HL-LHC resource requirements.

References

[1] *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3**, S08003 (2008)
[2] P. Calafiura, J. Catmore, D. Costanzo, A. Di Girolamo, *ATLAS HL-LHC Computing Conceptual Design Report*, Tech. Rep. CERN-LHCC-2020-015. LHCC-G-178, CERN, Geneva (2020), `https://cds.cern.ch/record/2729668`
[3] *Scikit-hep project*, `https://scikit-hep.org/` (2021), accessed: 2021-01-25

[4] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. Acosta, A. Melo, S. Belforte et al., *Coffea Columnar Object Framework For Effective Analysis*, EPJ Web Conf. **245**, 06012 (2020)

[5] J. Pivarski, P. Elmer, D. Lange, *Awkward Arrays in Python, C++, and Numba*, EPJ Web Conf. **245**, 05023 (2020), 2001.06307

[6] J. Elmsheuser et al., *Evolution of the ATLAS analysis model for Run-3 and prospects for HL-LHC*, EPJ Web Conf. **245**, 06014 (2020)

[7] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, in *Proceedings AIHENP'96 Workshop, Lausanne* (Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86., 1996)

[8] *Uproot*, https://github.com/scikit-hep/uproot4 (2021), accessed: 2021-01-26

[9] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., *Array programming with NumPy*, Nature **585**, 357 (2020)

[10] S.K. Lam, A. Pitrou, S. Seibert, *Numba: A LLVM-Based Python JIT Compiler*, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Association for Computing Machinery, New York, NY, USA, 2015), LLVM '15, ISBN 9781450340052, https://doi.org/10.1145/2833157.2833162

[11] *Apache parquet*, http://parquet.apache.org/ (2021), accessed: 2021-01-23

[12] The HDF Group, *Hierarchical Data Format, version 5*, https://www.hdfgroup.org/HDF5/ (1997-2021), accessed: 2021-01-25

[13] *NEP 1 — a simple file format for numpy arrays*, https://numpy.org/neps/nep-0001-npy-format.html (2021), accessed: 2021-01-26

[14] https://www.json.org (2021), accessed: 2021-01-26

[15] J. Blomer, P. Canal, A. Naumann, D. Piparo, *Evolution of the ROOT Tree I/O*, EPJ Web Conf. **245**, 02030 (2020)

[16] K. Ehatäht, *NANOAOD: a new compact event data format in CMS*, EPJ Web Conf. **245**, 06002 (2020)

[17] CMS Collaboration, *The CMS experiment at the CERN LHC*, Journal of Instrumentation **3**, S08004 (2008)

[18] ATLAS Collaboration, *Athena* (2019), https://doi.org/10.5281/zenodo.3932810

[19] *SUSYTools Athena package*, https://gitlab.cern.ch/atlas/athena/-/tree/21.2/PhysicsAnalysis/SUSYPhys/SUSYTools (2021), accessed: 2021-02-23