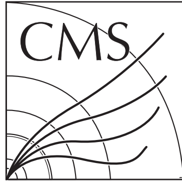
Available on CMS information server

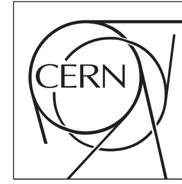
CMS CR -2022/041



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



02 March 2022 (v3, 31 March 2022)

Evaluating awkward arrays, uproot, and coffea as a query platform for High Energy Physics data

Nick Smith, Lindsey Gray

Abstract

Query languages for High Energy Physics (HEP) are an ever present topic within the field. A query language that can efficiently represent the nested data structures that encode the statistical and physical meaning of HEP data will help analysts by ensuring their code is more clear and pertinent. As the result of a multi-year effort to develop an in-memory columnar representation of high energy physics data, the numpy, awkward arrays, and uproot python packages present a mature and efficient interface to HEP data. Atop that base, the coffea package adds functionality to launch queries at scale, manage and apply experiment-specific transformations to data, and present a rich object-oriented columnar data representation to the analyst. Recently, a set of Analysis Description Language (ADL) benchmarks has been established to compare HEP queries in multiple languages and frameworks. In this paper we present these benchmark queries implemented within the coffea framework and discuss their readability and performance characteristics. We find that the columnar queries perform as well or better than the implementations given in previous studies.

Presented at *ACAT2021 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*

Evaluating Awkward Arrays, uproot, and coffea as a query platform for High Energy Physics Data

L. Gray, N. Smith, FNAL, Batavia, IL, 60510
On behalf of the CMS Collaboration

Abstract. Query languages for High Energy Physics (HEP) are an ever present topic within the field. A query language that can efficiently represent the nested data structures that encode the statistical and physical meaning of HEP data will help analysts by ensuring their code is more clear and pertinent. As the result of a multi-year effort to develop an in-memory columnar representation of high energy physics data, the NumPy, Awkward Array, and uproot python packages present a mature and efficient interface to HEP data. Atop that base, the coffea package adds functionality to launch queries at scale, manage and apply experiment-specific transformations to data, and present a rich object-oriented columnar data representation to the analyst. Recently, a set of Analysis Description Language (ADL) benchmarks has been established to compare HEP queries in multiple languages and frameworks. In this paper we present these benchmark queries implemented within the coffea framework and discuss their readability and performance characteristics. We find that the columnar queries perform as well or better than the implementations given in previous studies.

1. Introduction

Data in HEP is most often already stored on disk as a columnar data format within ROOT [1] files. For physics analysis the data typically occur either once per event, "event quantities", or zero or more times per event depending on the presence of some interesting detector signature, "physics-object" quantities. Both of these common patterns have efficient columnar representations on disk [2] and in memory [3, 4, 5] . Most common interfaces to ROOT files expose an interface where a single event is loaded into memory and then processed at a time. For data analysis, where most operations are selections on event and physics-object quantities, this prevents usage of processor vector instructions across event boundaries.

To take advantage of this we adopt the notation used in NumPy [4] to manipulate tensors of data, with extensions to deal with manipulations of nested arrays easily provided in the python package Awkward Array [6]. Similar to NumPy, Awkward Array is designed to deal with nested-array data structures (as opposed to rectangular arrays of data with NumPy) using an "array programming" approach where a given action is applied to all elements of an array using underlying C++ to execute the operation rather than an explicit for-loop in python. This avoids numerous time consuming aspects of loops and variable tracking in the python language, and switching from python loops to array programming can result in orders-of-magnitude reductions in runtime [6]. This effectively merges the flexibility of the python programming language with the efficiency of compiled languages. Furthermore, this approach directly exposes optimizations across events since only whole arrays are considered at a time, and results in on-disk and in-memory data layouts being closer to each other. The uproot [7] package manages reading

columnar data from ROOT files into in-memory arrays and presents an interface that provides access to ROOT files by a number of common protocols.

With these two packages it is possible to explore HEP data in a very similar way to using NumPy or pandas [4, 5] in the context of other areas of big data. However, a number of things are missing in order to develop complete high energy physics analyses using these tools. The coffea [8] package contains additions to manage and apply correction data of various kinds, a facility for horizontally scaling embarrassingly parallel task queues, and an object oriented interface to manage and query columnar data. Coffea is currently in use predominantly within the CMS Collaboration [9], by roughly 40 analyses. The latter two pieces of coffea will be used most prominently in this work since they comprise the user interface of constructing and evaluating queries in this software ecosystem.

In order to benchmark this software package we use an agreed-upon common set of queries [10] that were used to classify the expressiveness of a given query model by measuring the number of characters and lines. These benchmarks are also good for understanding computational performance as they include a range of query complexities: simple selections, combinatorics, sorting, and four-vector math. One difference with respect to a realistic analysis workload is the amount of data removed by cuts before heavy processing. A detailed discussion of the query complexity is given in [11], as well as implementation details in a variety of languages. Here we will report the implementation details of the Awkward Array versions of these queries and provide enough information for easy comparison to the wider array of benchmarks in [11]. We find that from the user-experience and performance standpoints this approach compares favorably to other studied query systems.

2. Coffea Framework

The queries evaluated in this work are written using the coffea framework. Coffea knits together packages available within the scientific python ecosystem [12, 13] and the Scikit-HEP domain-specific ecosystem [14] to provide a cohesive user interface for performing data analysis in HEP. The primary general-purpose packages supporting coffea are: NumPy [4, 15] for vectorized math on arrays; Awkward Array [6] for handling non-rectangular and structured data; Numba [16] for accelerating selected custom math kernels; and Matplotlib [17] for visualization. The primary domain-specific packages supporting coffea are: Uproot [7] for reading data serialized in ROOT files; hist [18] a python binding and user interface to the boost-histogram [19] header-only C++ library; and mplhep [20] interface for HEP-specific visualizations. A summary of the dependencies among these packages is shown in Fig. 1.

By using these packages coffea implements columnar tools for common operations in HEP data analysis, which are generally binned look–up tables of values or functions. Coffea also provides a low boilerplate interface for launching horizontal scale–out of analyses using columnar packages and tools. Having all these tools knit together provides a low-overhead platform for developing and completing HEP analyses while reusing the common and efficient code–base provided by the scientific python community and SciKit-HEP.

A unique contribution of coffea is its NanoEvents subpackage, an object oriented framework for assembling unstructured, but related, arrays of data into common physics analysis concepts (e.g. muons, electrons, jets, etc.). This representation of columnar data is powerful because it makes columnar structures familiar and intuitive to manipulate. NanoEvents is also extensible since it separates the data source from the description of the data, and thus replacing input file formats (e.g. ROOT with Parquet) with the same internal naming conventions is easy. Similarly, since NanoEvents forms objects by defining patterns to generate relationships between columns it is easy to extend to the schema of other analysis frameworks and other experiments [21]. Finally, it is responsive and fast for programming and prototyping since it reads in data as late as possible to perform operations, as well as forwarding the underlying Awkward Array caching

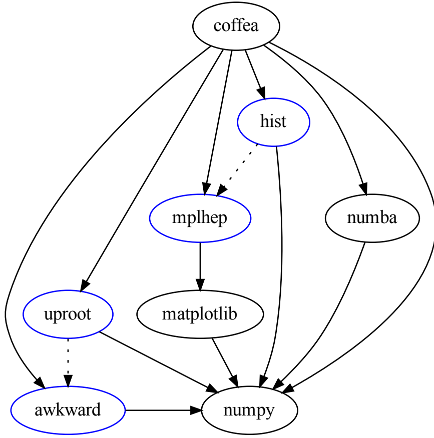


Figure 1. Dependencies among the primary software packages used in this work. Dashed lines indicate optional dependencies. Blue borders indicate packages that are part of the Scikit-HEP ecosystem [14].

interface to the user.

3. Query Implementation

The ADL benchmark queries [10] are a set of 8 simplified HEP analysis tasks representative of common data manipulation operations, designed to benchmark analysis language syntax. They are also useful to compare performance and scaling characteristics of analysis frameworks. Such a comparison was performed in Ref. [11] for RDataFrame [22] and industry query solutions. Here we perform a similar analysis of the coffea implementation of the tasks. The query tasks (labelled Q1-Q8) access from 1 to 14 data columns of a 17.3 GB input data file in the ROOT file format. The line and character counts of the coffea implementation[1] for each query, as well as the "boilerplate", i.e. code lines common to all queries, is listed in Table 1. Line counts are a subjective measurement, and to control somewhat this variance, the code is formatted with the well-known Python source code formatting tool `black` with its default arguments.

Table 1. Query implementation statistics

Query	Lines	Characters
Boilerplate	24	556
Q1	36	936
Q2	36	928
Q3	36	970
Q4	39	1073
Q5	47	1366
Q6	59	1905
Q7	52	1507
Q8	71	2462

[1] Viewable at https://github.com/CoffeaTeam/coffea-benchmarks/blob/ACAT2021/coffea-adl-benchmarks.ipynb

4. Performance Evaluation

The performance of the queries is evaluated using a Google Cloud Platform (GCP) `n2-standard-48` virtual machine. The machine has: 48 virtual CPU cores with Cascade Lake architecture, with hyperthreading enabled; 192 GiB of system memory; and a solid-state disk (SSD). The initial evaluation was to include benchmarks reading the input file from SSD as well as system memory (i.e. `/dev/shm`), however we found an unexpectedly low read speed for the SSD of only 50 MB/s, and decided to only analyze queries from memory. As the input file is compressed, the memory allocations required to decompress and deserialize the input file into data arrays are still taken into account. All evaluations use the coffea `FuturesExecutor` for multi-core tests and the `IterativeExecutor` for single-core tests. Pre-compiled binary packages made available through conda-forge were used for all evaluations.

4.1. Profiling and optimization

The initial software setup used `coffea` v0.7.5, `awkward` v1.4.0, and `uproot` v4.0.8. After profiling the benchmarks, we identified a few minor optimizations in the Awkward Array and coffea codebases, along with a user-facing optimization in Q6.

In Awkward Array, a common operation is to slice an array by an integer index array, similar to the `numpy.take` operation or fancy indexing. In profiling, these slice kernels took the second-most exclusive time after decompression of the input buffers. As they are essentially a sequence of many small `memcpy` operations, a specialization on the most common sizes (4, 8 bytes) avoids the `memcpy` function call, which improved the performance by factors up to 3x.

Most of the benchmark CPU time in the coffea framework is spent in Lorentz vector math. Since all math is expressed in interpreted code as a sequence of NumPy kernel calls, some optimizations are not automatic as they would be in compiled code, e.g. fusing the add and multiply operations in the invariant mass calculation $m^2 = E^2 - p_x^2 - p_y^2 - p_z^2$. Rewriting these operations as numba kernels led to a 20% overall improvement in the benchmarks.

For the remainder of the benchmarks, `coffea` v0.7.9 and `awkward` v1.5.1 were used, which included the improvements discussed in this section.

In Q6, a significant performance improvement is found by making a user-facing decision to pre-convert a Lorentz vector quantity from one coordinate system to another. The (p_T, η, ϕ, m) coordinate system is usually preferred for on-disk formats as the quantities most commonly filtered on are in this basis (p_T and η) and a common operation is to compute the pairwise distance between two vectors in the (η, ϕ) plane. However, for adding Lorentz vectors, it is much more efficient to represent them in cartesian coordinates. Since Q6 involves computing the vector sum of all combinations of three jets per event, in Listing 1 we repeat the conversion to cartesian coordinates for each combination of three jets. In Listing 2 we pre-convert the `events.Jet` array by assembling a new record structure with `ak.zip` from the derived quantities x, y, z, t computed on-demand at attribute access (`getattr`) and save them as the new array `jets`. The `btag` sub-array used later is included in the new array by reference as it is not a derived quantity.

Listing 1. Original Q6 selection

```
trijet = ak.combinations(
    events.Jet, 3, fields=["j1", "j2", "j3"]
)
trijet["p4"] = trijet.j1 + trijet.j2 + trijet.j3
```

Listing 2. Improved Q6 selection

```
jets = ak.zip(
    {
        k: getattr(events.Jet, k)
        for k in ["x", "y", "z", "t", "btag"]
    },
    with_name="LorentzVector",
    behavior=events.Jet.behavior,
)
trijet = ak.combinations(
    jets, 3, fields=["j1", "j2", "j3"]
)
trijet["p4"] = trijet.j1 + trijet.j2 + trijet.j3
```

At present, all operations shown are eager. A core goal of Awkward Array v2 is to enable construction of a task graph for delayed computation. With that graph in hand, it may be possible to detect situations like these and optimize them automatically.

4.2. Task Splitting Evaluation

Coffea parallelizes the processing intra-file by dividing it into equal-length *chunks* of event records where the number of records is chosen such that the length is as close as possible to a target chunk size. The target chunk size is currently a user-facing parameter, although the `WorkQueueExecutor` includes a automatic chunk size optimization option. As seen in Fig. 3, the lower the chunk size, the more time is spent in python overhead compared to the vectorized operations on contiguous arrays of data. In addition, for each chunk, the file is re-parsed to find the appropriate byte offsets and extract the data, which increases the total amount of bytes read as shown in Fig. 2. This could be improved if coffea serialized and sent the file metadata to the workers, however in current practice, having a sufficiently large chunk size makes this overhead tolerable.

The upper bound on chunk size is the amount of memory required to hold all the input and intermediate value arrays for a chunk. After a point, this memory pressure slows the application, as seen in Fig. 3.

For the rest of the evaluation, we fix the target chunk size parameter to 2^{19} events, as it is optimal for the most expensive queries, namely Q6 and Q7. The coffea default target chunk size is 100k events.

4.3. Result Aggregation Evaluation

All coffea executors collect and merge the worker outputs in the client process with the exception of `DaskExecutor`, which performs a tree reduction in the workers. For the `FuturesExecutor`, we check whether the merging step is limiting performance by comparing the CPU time spent in the client process versus the chunk throughput for various chunk sizes, worker counts, and output sizes. The output sizes are varied by modifying Q2 to produce 100-bin 1-dimensional, 10k-bin 2D, and 1M-bin 3D jet kinematics (p_T, η, ϕ) histograms. In Fig. 4, we see that even for the largest output size and highest chunk throughput, the client process never reaches full CPU utilization.

We also tried the same benchmark with the `DaskExecutor` but found lower overall performance compared to the `FuturesExecutor` (which is using the Python standard library `concurrent.futures.ProcessPoolExecutor`) on a single machine.

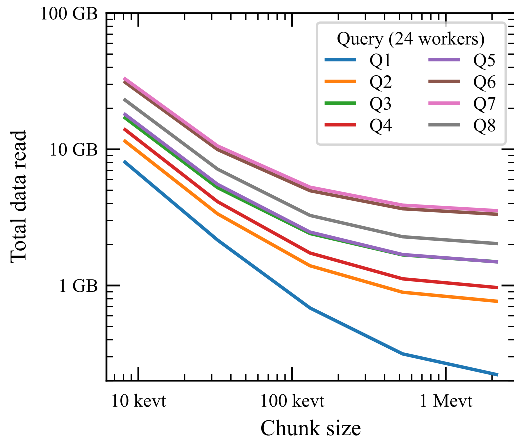


Figure 2. Total bytes read from the input file during processing as a function of chunk size. The input file is 17.3 GB in size.

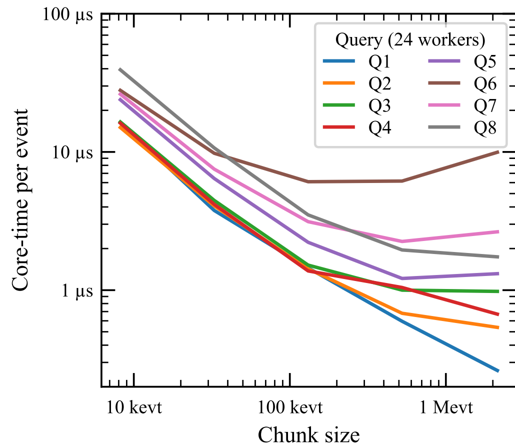


Figure 3. Core-time per event as a function of chunk size. Core-time per event is defined as the wall-clock processing time multiplied by the number of worker processes divided by the number of event records in the input file.

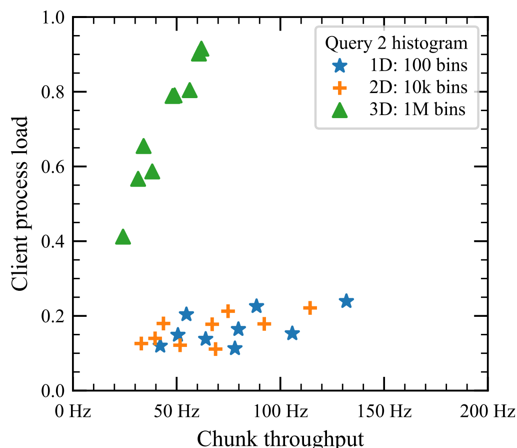


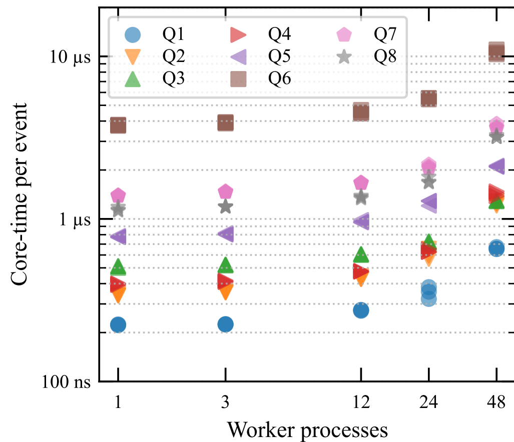
Figure 4. Client process load versus chunk throughput for various chunk sizes, worker counts, and output sizes. Process load is defined as user plus system time divided by wall-clock time. Chunk throughput is defined as the number of chunks processed divided by wall-clock time.

4.4. Scaling Evaluation

In Fig. 5 we explore the scaling of the system by varying the number of worker processes employed by `FuturesExecutor`, and find that the scaling is not ideal. We suspect the strong inflection going fro 24 to 48 processes has to do with the fact that GCP provisions by the hyperthread rather than by real cores.

To ensure any variation is not due to external factors (e.g. co-located VMs) we re-ran the benchmarks 3 times on two different GCP VM instances. Fig. 6 shows little variation in the results of this repeatability test.

In Fig. 7 we compare the performance with that of a pre-compiled C++ RDataFrame implementation of the same queries, using ROOT v6.24.6 installed via conda-forge. The same coordinate system transform optimization for Q6 was also implemented in RDataFrame thanks to the help of ROOT experts. For both analysis frameworks, some non-ideal scaling is observed. RDataFrame has equal or better single-thread performance in all queries, while the scaling behavior of the two frameworks varies by query. Future work to understand the cause of the



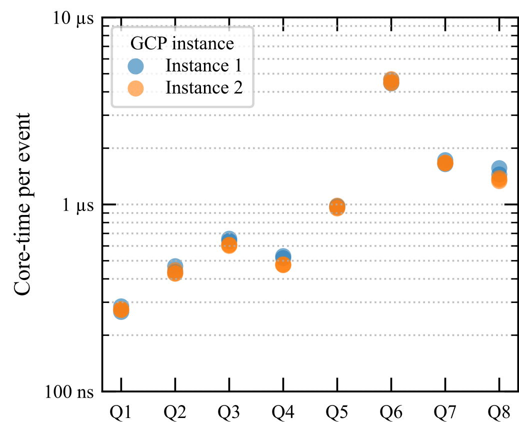


Figure 5. Core-time per event as a function of worker process count. Each query is repated three time per worker process setting. The horizontal guide lines represent ideal scaling behavior.

Figure 6. Core-time per event for repeated queries on different GCP instances.

differences will likely be useful to both frameworks' developers.

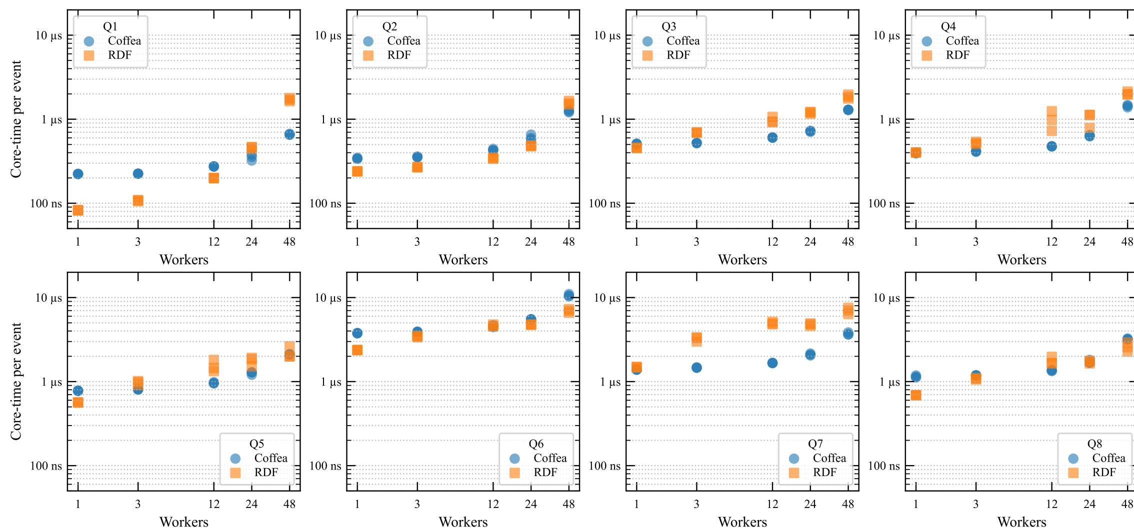


Figure 7. Core-time per event as a function of worker count. Core-time per event is defined as the wall-clock processing time multiplied by the number of worker processes divided by the number of event records in the input file. For RDataFrame (RDF), the workers are threads, while for coffea the workers are child processes.

5. Conclusion

In this study we presented a fair and well-controlled comparison of the coffea and ROOT analysis frameworks using well-known and publicly agreed upon analysis examples. We performed a brief study of the query-language characteristics of Awkward Array in the context of the coffea

framework, for comparison to other studies done by Graur et al [11]. On a small data sample with a range of computational loads it is found that the columnar and ROOT-based approaches perform similarly, with ROOT typically having a slight advantage, and further studies are needed to understand contiguous runs on larger in-memory datasets. The scaling properties vs. core-multiplicity of the two methodologies are studied as well and demonstrating subtle differences coming from threading and multiprocessing models as well as differing overheads for process initialization. Since the outcome of these studies show that the performance of these two techniques is very similar, further testing will be done to understand scaling in real physics analysis scenarios where orders of magnitude more data are required.

Acknowledgments
This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. We also thank Jim Pivarski for maintaining Awkward Array and uproot, and Enrico Guiraud for discussions which helped to make the comparisons in this work as fair as possible.

[1] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth. A*, 389:81–86, 1997.
[2] Quincey Koziol. *HDF5*, pages 827–833. Springer US, Boston, MA, 2011.
[3] Geoffrey Lentner. Shared memory high throughput computing with apache arrow™. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.
[4] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.
[5] Wes McKinney. Data structures for statistical computing in python. pages 56–61, 01 2010.
[6] Jim Pivarski, Peter Elmer, and David Lange. Awkward Arrays in Python, C++, and Numba. *EPJ Web Conf.*, 245:05023, 2020.
[7] Jim Pivarski, Henry Schreiner, Nicholas Smith, et al. scikit-hep/uproot4: 4.2.1, March 2022.
[8] Nicholas Smith et al. Coffea: Columnar Object Framework For Effective Analysis. *EPJ Web Conf.*, 245:06012, 2020.
[9] S. Chatrchyan et al. The CMS Experiment at the CERN LHC. *JINST*, 3:S08004, 2008.
[10] Mason Proffitt, Ingo Müller, Mat Adamec, Pieter David, Enrico Guiraud, and Sebastien Binet. iris-hep/adl-benchmarks-index: ADL Functionality Benchmarks Index v0.1, July 2021.
[11] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. Evaluating Query Languages and Systems for High-Energy Physics Data. *Proc. VLDB Endow.*, 15(2), 4 2021.
[12] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.
[13] K. Jarrod Millman and Michael Aivazis. Python for Scientists and Engineers. *Comput. Sci. Eng.*, 13(2):9–12, 2011.
[14] Eduardo Rodrigues et al. The Scikit HEP Project – overview and prospects. *EPJ Web Conf.*, 245:06028, 2020.
[15] Charles R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.
[16] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.
[17] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
[18] Henry Schreiner, N!no, Aman Goel, et al. scikit-hep/hist: Version 2.6.0, February 2022.
[19] Henry Schreiner, Hans Dembinski, N!no, et al. scikit-hep/boost-histogram: Version 1.3.1, February 2022.
[20] Andrzej Novak, Henry Schreiner, Matthew Feickert, et al. scikit-hep/mplhep: v0.3.23, March 2022.
[21] Hartmann, Nikolai, Elmsheuser, Johannes, Duckeck, Günter, and on behalf of ATLAS Software and Computing. Columnar data analysis with atlas analysis formats. *EPJ Web Conf.*, 251:03001, 2021.
[22] Vincenzo Eduardo Padulano, Javier Cervantes Villanueva, Enrico Guiraud, and Enric Tejedor Saavedra. Distributed data analysis with ROOT RDataFrame. *EPJ Web Conf.*, 245:03009, 2020.