Toward real-time data query systems in HEP

Jim Pivarski[1], **David Lange**[1], **Thanat Jatuphattharachat**[2]

[1] Physics Department, Princeton University, Princeton, NJ 08544, USA

[2] Computer Engineering Chulalongkorn University, Krung Thep Maha Nakhon 10330, Thailand

E-mail: `pivarski@princeton.edu`

Abstract. Exploratory data analysis tools must respond quickly to a user's questions, so that the answer to one question (e.g. a visualized histogram or fit) can influence the next. In some SQL-based query systems used in industry, even very large (petabyte) datasets can be summarized on a human timescale (seconds), employing techniques such as columnar data representation, caching, indexing, and code generation/JIT-compilation. This article describes progress toward realizing such a system for High Energy Physics (HEP), focusing on the intermediate problems of optimizing data access and calculations for "query sized" payloads, such as a single histogram or group of histograms, rather than large reconstruction or data-skimming jobs. These techniques include direct extraction of ROOT TBranches into Numpy arrays and compilation of Python analysis functions (rather than SQL) to be executed very quickly. We will also discuss the problem of caching and actively delivering jobs to worker nodes that have the necessary input data preloaded in cache. All of these pieces of the larger solution are available as standalone GitHub repositories, and could be used in current analyses.

1. Benefits of a query service

Despite everything that's changed in computing since the first AIHENP/ACAT workshop, one aspect of today's High Energy Physics (HEP) analyses that a physicist 27 years ago would recognize is the use of private skims. Rather than analyzing the reconstructed data directly, physicists skim it (dropping events) and slim it (dropping particle attributes), copy this summary closer to where they will be working, and then analyze the summary. The reason is performance: smaller data with fewer steps between the analyst and the data result in quicker exploratory plots, which allow the human analyst to be more engaged in the investigation. Long processing times are acceptable for established, push-button procedures, but a new analysis requires creative work and interactive discovery.

However, producing such a skim can take a long time— weeks or months before seeing the first plot— and is fraught with tradeoffs. Excluding important events or (more often) important particle attributes can be costly, and including unnecessary data only makes the process take longer and use more disk. Moreover, the copy uses extra resources (disk and processing), which will only get tighter as data volumes outpace computing budgets, and it can effectively "price out" small analysis groups. Since it is a distinct copy, it does not benefit from improvements or corrections to the source data until refreshed, and can often be a year or more out of date. Finally, it forces physicists to take time and attention away from physics and statistical issues to solve an IT problem. If the analysis can be performed without this skim, so much the better.

Table 1. Rate of processing a "query sized" payload: filling one histogram of jet p_T for all jets in a $t\bar{t}$ sample, illustrating the orders of magnitude lost to providing a full framework for heavy event processing (all single-threaded).

rate	operation
0.018 MHz	full framework (CMSSW, single-threaded C++)
0.029 MHz	load all 95 jet branches in ROOT
2.8 MHz	load jet p_T branch (and no others) in ROOT
12 MHz	allocate C++ objects on heap, fill, delete
31 MHz	allocate C++ objects on stack, fill histogram
250 MHz	minimal "for" loop in memory (single-threaded C)

Real-time analysis on primary datasets is possible, and has been demonstrated in industry, usually in the form of large, distributed databases used internally by a company's data analytics team. These systems are built from Apache Drill, Impala, Kudu, Hawk, or Dremel and tuned to provide SQL access to petabytes of data in seconds. Unlike Hadoop and Spark, which are optimized for high throughput, these systems are optimized for low latency, and they use database-style indexes and query planning to avoid churning through all data with every request.

We can learn from these systems, adopting their techniques to provide similar access to HEP data, but there are some important differences between industry needs and HEP needs. A physicist's analysis is both simpler and more complex than a data scientist's analysis: it is simpler in that HEP does not require all-to-all processing, such as transposing a table of customer purchases to make product recommendations— all of the information that is needed to process one physics event is contained in that event or small auxiliary tables, not all the other events. Once collected, a HEP dataset is immutable— new events do not accumulate while the physicist is analyzing it. However, physics analyses are more complex in the depth of processing applied to that one event, examining many combinations of particles to search for relationships among the particles. To help with the bookkeeping, physicists are inclined to think in terms of objects, iterating over collections of particle objects, rather than exploding and aggregating tables as in SQL.

Because of the differences, we do not expect an off-the-shelf database to solve our problem directly, but we can adapt established techniques to work for HEP.

2. Optimizing for "query sized" payloads

An analysis query is the smallest unit of exploratory analysis, in which the physicist asks one question, such as "What is the distribution of muon η for the second-highest p_T muon in each event, subject to track-fit χ^2 constraints?" The conditions may be very complex, involving mass calculations, vertex fits, generated/reconstructed particle matching, computed with deeply nested loops and external function calls, but it is a request for one plot. The request and the plot are small enough to be transmitted through a slow network, and the data processing touches at most a dozen particle attributes out of thousands. Even a petabyte-sized dataset would only need to yield a terabyte of data, distributed among hundreds of workers, to service this request.

HEP frameworks are not designed for this style of data access. Table 1 shows how four orders of magnitude in processing rate are lost to provide all the services of a full event framework (CMSSW in this example). These include creating full C++ particle objects with all attributes loaded, allocated randomly on the heap for convenient memory management, compared to a sequential, contiguous, and possibly vectorized histogram fill over a simple array. These services are essential for "heavy" processing tasks like event reconstruction, which touch most attributes of each particle, but are inappropriate for "query sized" tasks like filling a histogram.

Table 2. Illustration of exploding nested, hierarchical objects. The data below are logically conceived as a list of lists of character/integer pairs, but stored in memory as four flat arrays describing the structure of the outer list, the inner list, the 1st and 2nd attributes.

logical data	`[[(a,1),`	`(b,2),`	`(c,3),`	`(d,4)],`	`[],`	`[(e,5),`	`(f,6)]],`	`[],`	`[[(g,7)]]`	
outer offsets	`[0,`							`3,`	`3,`	`4]`
inner offsets	`[ 0,`				`4,`	`4,`			`6,`	`7]`
1st attribute	`[ a,`	`b,`	`c,`	`d,`		`e,`	`f,`		`g`	`]`
2nd attribute	`[ 1,`	`2,`	`3,`	`4,`		`5,`	`6,`		`7`	`]`

Private skims strip most of this away for lightweight processing in a physicist's personal analysis environment; there is no reason a lightweight framework could not also be provided centrally.

Similar to low-latency databases used in industry, ROOT stores objects in an "exploded" form (called "splitting" in ROOT), with all values of each attribute in a separate array. This is even possible for hierarchically nested data, containing arbitrary length lists of objects, though multiple arrays with different lengths are required to encode it. This allows ROOT to only read attributes required for the analysis function— such as a terabyte of a petabyte dataset.

However, databases like Apache Drill go one step further by leaving the data in their exploded form during processing, without materializing rows as runtime objects[1]. Selective reading accounts for the first two orders of magnitude in Table 1, while eliminating object materialization accounts for most of the last two by skipping memory allocation, non-sequential memory scans, and permitting compiler optimizations like loop vectorization.

To permit this kind of access for ROOT data, Brian Bockelman and Zhe Zhang added a method to ROOT I/O that skips the `GetEntry` (event materialization) process[2]. Query sized calculations on the resulting arrays (computing momentum magnitudes from components) run 5 times faster than a streamlined `GetEntry` loop and 10 times faster than `TTree::Draw` or `TTreeReader` for uncompressed or LZ4-compressed input data.

3. Code transformation and compilation

For interactive analysis, it must be possible to define analysis functions at runtime and then compile them to run at full speed on arrays. Drill converts SQL to Java bytecode[1], while Brian and Zhe's test cases use Numba to JIT-compile Python[2]. What these tools lack is a mechanism to convert a physicist's object oriented view of the data into array operations. Below, I will describe how we can do this and how the transformed code performs.

Given the exploded data shown in Table 2 and user code such as

```
for outerlist in dataset:
   for innerlist in outerlist:
      for pair in innerlist:
         compute(pair.first, pair.second)
```

we want to convert it to a form that doesn't reference "`dataset`," "`outerlist`," "`innerlist`," or "`pair`" as objects, instead deferring to "`outeroffsets`," "`inneroffsets`," "`first`" and "`second`," where the data are stored as arrays.

Such a transformation can be performed algorithmically on the user code's Abstract Syntax Tree (AST), converting the above into

```
for (i = 0;  i < 3;  i++)
   for (j = outeroffsets[i];  j < outeroffsets[i+1];  j++)
      for (k = inneroffsets[j];  k < inneroffsets[j+1];  k++)
         compute(first[k], second[k]);
```

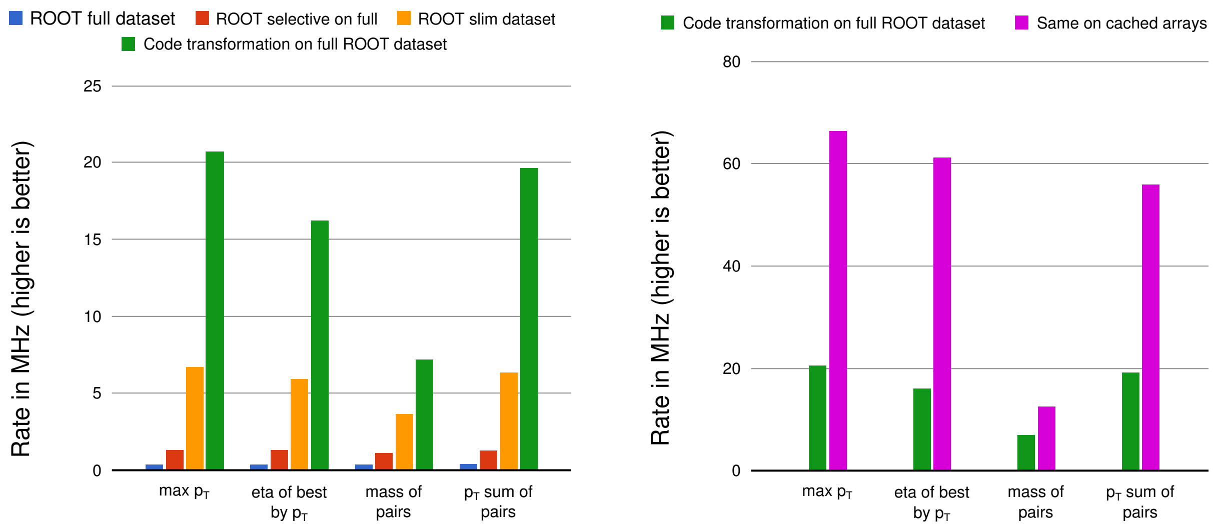

Figure 1. Processing rate of four analysis functions (Table 3) implemented in C++ with `GetEntry` and our Python code transformation/compilation. "ROOT full dataset" loads all particle attributes, including unnecessary ones, "selective on full" uses selective reading (`SetBranchStatus`), "slim dataset" uses a specially preselected input sample, "Code transformation on full ROOT dataset" uses Brian and Zhe's array extraction and our code transformation. Though uncompressed in warm cache, ROOT file reading is still the bottleneck; transformed code on raw arrays cached in memory is several times faster (all single-threaded).

by replacing each "`outerlist`" AST node with its corresponding "`outeroffsets[i]`" and each "`pair.first`" with its corresponding "`first[k]`." The first transformation rule eliminates references to list objects and the second eliminates references to record objects.

This transformation is like a type-inferring compilation pass, in which the types of dataset substructures must be propagated through the code, including assignments to new variables, and syntactic structures such as "for" loops have special transformation rules. Functions such as "len" (requesting the length of a list) must be overloaded as well (e.g. "`offsets[j+1] - offsets[j]`").

In special cases such as this one, we can even convert the above into

```
for (k = 0;  k < inner[outer[3]];  k++)
   compute(first[k], second[k]);
```

by recognizing that the outer and inner loops are total and sequential. Thus, when the transformed function is passed to a bytecode compiler such as Numba or Clang, the non-nested "for" loop may be more highly optimized, possibly vectorized (depending on "`compute`"). This code transformation technique is discussed in more detail in[3].

The performance of the transformed code was studied with a simulated Drell-Yan dataset containing 5.4 million collisions in the CMS detector. Calculations were performed in single-threaded processes on a `i2.xlarge` Amazon Web Services (AWS) instance, files were read from warmed cache, and the contents were uncompressed. Results are presented in Figure 1.

4. Distributed processing with cache

The optimizations discussed above focus on increasing single-threaded throughput, but of course the main benefit of a centralized query service is parallelization. If we attain our latency goal of no more than a second per plot and a hundred physicists are online, submitting a query every ten seconds, then each physicist would get a tenth of the whole cluster at a time.

Table 3. Analysis functions tested in Figure 1: "max p_T" is a per-event aggregation, "eta of best" adds the complication of maximizing one attribute while plotting another, "mass of pairs" iterates over distinct pairs of particles per event, computing an essential HEP function, and "p_T sum of pairs" has the same loop structure without the expensive cosh and cos calls.

max $\mathbf{p_T}$

```
for event in dataset:
  maximum = 0.0
  for muon in event.Muon:
    if muon.pt > maximum:
      maximum = muon.pt
  fill_histogram(maximum)
```

eta of best by $\mathbf{p_T}$

```
for event in dataset:
  maximum = 0.0
  best = None
  for muon in event.muons:
    if muon.pt > maximum:
      maximum = muon.pt
      best = muon
  if best is not None:
    fill_histogram(best.eta)
```

mass of pairs

```
for event in dataset:
  n = len(event.muons)
  for i in range(n):
    for j in range(i+1, n):
      m1 = event.muons[i]
      m2 = event.muons[j]
      mass = sqrt(
        2*m1.pt*m2.pt*(
        cosh(m1.eta - m2.eta) -
        cos(m1.phi - m2.phi)))
      fill_histogram(mass)
```

$\mathbf{p_T}$ sum of pairs

```
for event in dataset:
  n = len(event.muons)
  for i in range(n):
    for j in range(i+1, n):
      m1 = event.muons[i]
      m2 = event.muons[j]
      s = m1.pt + m2.pt
      fill_histogram(s)
```

HEP analysis functions are embarrasingly parallel, which would scale linearly if data can be delivered to them efficiently. However, this depends critically on caching, since data processing is considerably faster than disk access, and caching is local. An input dataset in memory on one machine is only useful if subsequent jobs requiring that input are sent to the same machine.

Rather than dispatch subtasks round-robin or to the least busy compute node, we want compute nodes to pull subtasks with a preference for input data they already have in cache. Figure 2 shows a scheme for accomplishing this, using Apache Zookeeper to advertise new subtasks and globally mark them as in progress and delete them when done. Compute nodes pull subtasks rather than having work pushed to them because only the compute nodes know which input datasets are locally in cache. Pulling work also reduces the effect of stragglers for the same reason as work-stealing. Additionally, the work-pulling algorithm should not strictly

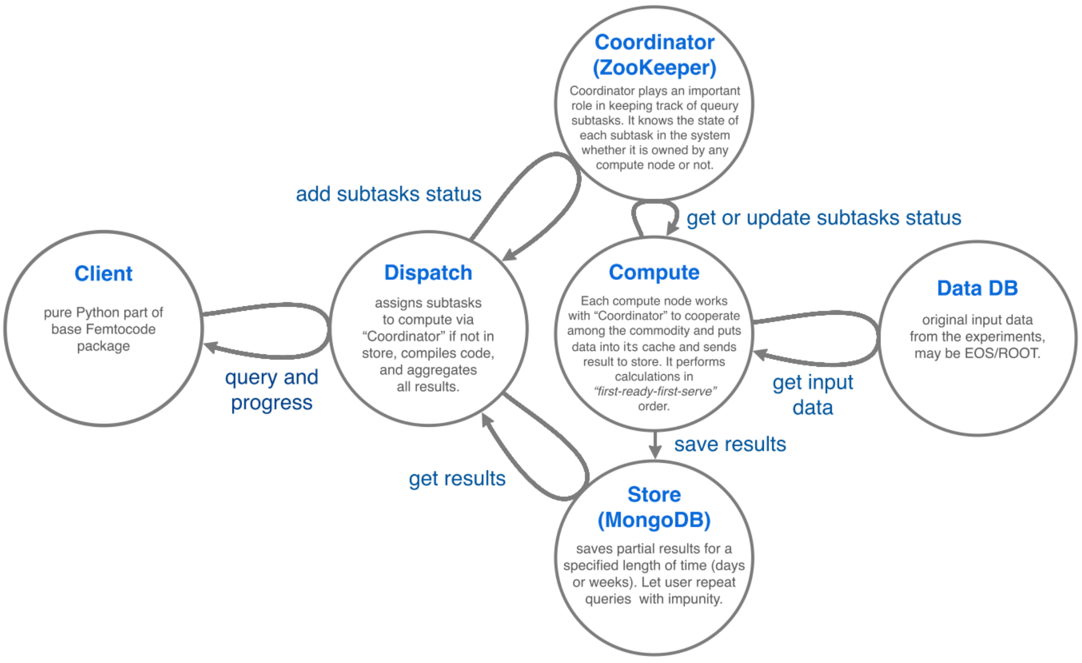
Coordinator (ZooKeeper)
Coordinator plays an important role in keeping track of queury subtasks. It knows the state of each subtask in the system whether it is owned by any compute node or not.
add subtasks status
get or update subtasks status
Client
pure Python part of base Femtocode package
query and progress
Dispatch
assigns subtasks to compute via "Coordinator" if not in store, compiles code, and aggregates all results.
Compute
Each compute node works with "Coordinator" to cooperate among the commodity and puts data into it's cache and sends result to store. It performs calculations in "first-ready-first-serve" order.
Data DB
original input data from the experiments, may be EOS/ROOT.
get input data
save results
get results
Store (MongoDB)
saves partial results for a specified length of time (days or weeks). Let user repeat queries with impunity.

Figure 2. Schematic for distributed query processing to minimize cache misses (see text).

require input data to be in cache, but should check for work in two rounds: the first takes only cache-local work, but if there is no cache-local work to do, compute nodes will take any work after a sub-second delay. This allows the compute nodes that can best service the request to have first dibs, but if there's too much for them to do, new compute nodes will download the input data so that the handling of a popular dataset elastically scales with demand.

For the most common type of query, histogram aggregation, subtasks represent partial histograms that must be aggregated centrally. For responsiveness, we imagine storing partial histograms in a document database like MongoDB and aggregating whatever is available at regular intervals. That way, the user would see results accumulate interactively and can cancel malformed queries. The range of supported tasks can be extended by adopting generalized aggregation with Histogrammar[4].

This system was developed in draft form by one of us (Thanat)[5], to serve as a model for development. This draft system lacks only realistic work payloads, substituting them with random delays.

5. Summary

A centralized query service that is as fast and convenient as private skims would have a profound impact on the working habits of thousands of physicists. It doesn't represent an incremental change of existing infrastructures, and as such, needs quite a few new components to make it work.

Rather than trying to introduce this system in a "big bang," we are rolling out individual components, even if they tend to increase reliance on private skims in the short term. For instance, array access to ROOT data and Python code transformation/compilation can accelerate and simplify analysis using skims. We are considering ways of implementing a distributed system as a virtual ROOT file, to provide physicists with a familiar interface, even if it isn't used with the accelerated query engine. This way, we can get feedback about parts of the system before the whole is in place. This method also encourages spin-off technologies.

The following software is either available now or soon.

- Array access to ROOT data, via Numpy and `TTreeReaderFast`, is scheduled for **ROOT 6.14**[6]. Users willing to compile ROOT may try the feature early using a custom fork[7].
- The same interface is available now in a Python package called **uproot**[8].
- The code transformation techniques described here are implemented in **OAMap**[9], which accelerates functions on data in Apache Arrow[10] format, with on-the-fly conversion from ROOT to Arrow. The purpose of targeting Arrow rather than ROOT directly is to attract contributions from Big Data projects that rely on Arrow.

Readers are encouraged to try out these packages and report any usability issues, as this will guide the process of developing the query system.

Acknowledgements

This work was supported by the National Science Foundation under grants ACI-1450377 and PHY-1624356.

References

[1] M Hausenblas and J Nadeau, "Apache drill: Interactive ad-hoc analysis at scale," *Big Data*, vol. 1(2), pp. 100–104, 2013. [Online]. Available: https://doi.org/10.1089/big.2013.0011

[2] B Bockelman, Z Zhang, and J Pivarski, "ROOT-BulkIO and Numpy interface," *Journal of Physics: Conference Series*, 2017. [Online]. Available: https://indico.cern.ch/event/567550/contributions/2627167/

[3] J Pivarski, P Elmer, B Bockelman, Z Zhang, "Fast Access to Columnar, Hierarchical Data via Code Transformation," *http://arxiv.org/abs/1708.08319*, Aug. 2017.

[4] J Pivarski, A Svyatkovskiy, F Schenck, and B Engels, “histogrammar-python: 1.0.0,” Sep. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.61418

[5] T Jatuphattharachat. (2017) femto-mesos. [Online]. Available: https://github.com/JThanat/femto-mesos

[6] R Brun and F Rademakers, “ROOT: An object oriented data analysis framework,” *Nucl. Instrum. Meth.*, vol. A389, pp. 81–86, 1997. [Online]. Available: https://doi.org/10.1016/S0168-9002(97)00048-X

[7] B Bockelman. (2017) root-bulkapi-fastread-v2. [Online]. Available: http://github.com/bbockelm/root/tree/root-bulkapi-fastread-v2

[8] J Pivarski. (2017) “Uproot: Minimalist ROOT I/O in pure Python and Numpy.”. [Online]. Available: https://github.com/scikit-hep/uproot

[9] ——. (2017) “OAMap: Toolset for Computing Directly on Hierarchically Nested, Columnar Data”. [Online]. Available: https://github.com/diana-hep/oamap

[10] The Apache Arrow team. (2016) “Apache Arrow: Powering Columnar In-Memory Analytics”. [Online]. Available: https://arrow.apache.org/