Toward real-time data query systems in HEP

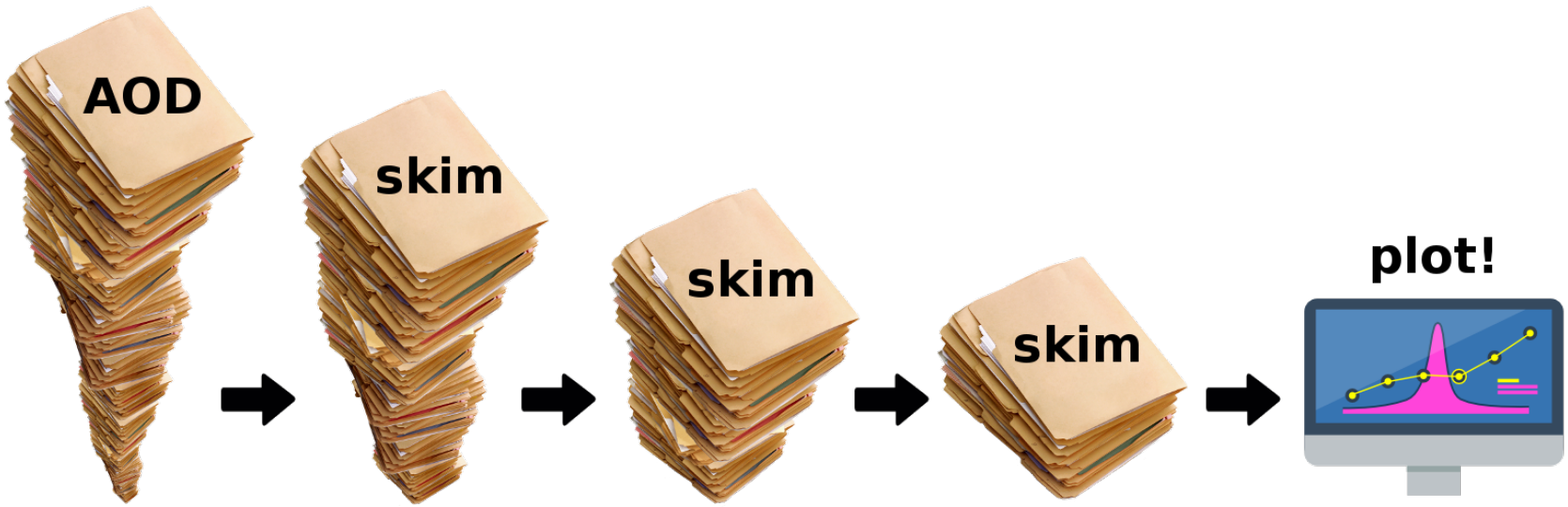
Jim Pivarski

Princeton University – DIANA

August 22, 2017

The dream...

LHC experiments centrally manage data from readout to Analysis Object Datasets (AODs), but the next stage, from AODs to final plots, is handled separately by each physics group, often copying the data several times.



The dream...

LHC experiments centrally manage data from readout to Analysis Object Datasets (AODs), but the next stage, from AODs to final plots, is handled separately by each physics group, often copying the data several times.

Wouldn't it be nice if physicists could directly turn AOD into plots, quickly enough for interactive analysis (seconds per scan at most)?

Why this would be a Good Thing

- Users could plot the data before deciding what to keep.

  (Even if they do need skims for maximum likelihood fits, etc, these can be better streamlined *after* exploratory plotting.)

- Focus on physics and statistical issues, not data handling.

- Centralization facilitates provenance and reproducibility.

- Shared CPU, disk, and memory can be more efficient.

- Small institutions would not be "priced out" of analysis for lack of resources to copy and locally process the data.

Existence proof

In some industries, it is possible to "process petabyes of data and trillions of records in seconds[1]," usually as SQL.

In fact, there are many low-latency query server engines available, mostly open source.

[1] https://wiki.apache.org/incubator/DrillProposal

Existence proof

In some industries, it is possible to "process petabyes of data and trillions of records in seconds[1]," usually as SQL.

In fact, there are many low-latency query server engines available, mostly open source.

Apache Drill comes closest to fitting our needs, but it's

- SQL (not expressive enough for HEP)
- Java (hard to link to HEP software)
- more suited to "flat ntuple" analysis (see next slides).

[1] https://wiki.apache.org/incubator/DrillProposal

Toward a query system for HEP

This talk is about what we would need to make (or alter) a query system for HEP analysis:

- **fast execution** on *nested, non-flat* data
- **distributed processing:** caching and data locality, uneven and changing dataset popularity, aggregation
- ~~**HEP-specific query language**~~ (in the abstract for this talk, but I'm going to focus more on the above)

Fast execution

The core issue

SQL-like query engines are optimized for what we'd call a "flat ntuple analysis"— rectangular table of numbers, sliced, filtered, aggregated, and joined.

The core issue

SQL-like query engines are optimized for what we'd call a "flat ntuple analysis"— rectangular table of numbers, sliced, filtered, aggregated, and joined.

Only some late-stage HEP analyses fit this model, not AOD.

The core issue

SQL-like query engines are optimized for what we'd call a "flat ntuple analysis"— rectangular table of numbers, sliced, filtered, aggregated, and joined.

Only some late-stage HEP analyses fit this model, not AOD.

In general, physics analysis requires arbitrary-length lists of objects: e.g. events containing jets containing tracks containing hits.

The core issue

SQL-like query engines are optimized for what we'd call a "flat ntuple analysis"— rectangular table of numbers, sliced, filtered, aggregated, and joined.

Only some late-stage HEP analyses fit this model, not AOD.

In general, physics analysis requires arbitrary-length lists of objects: e.g. events containing jets containing tracks containing hits.

But frameworks that create physics objects at runtime would be slow to process as queries.

Illustration with a microbenchmark

Query: fill a histogram with jet p_T of all jets.

0.018 MHz full framework (CMSSW, single-threaded C++)

250 MHz minimal "for" loop in memory (single-threaded C)

Illustration with a microbenchmark

Query: fill a histogram with jet p_T of all jets.

0.018 MHz	full framework (CMSSW, single-threaded C++)
31 MHz	allocate C++ objects on stack, fill histogram
250 MHz	minimal "for" loop in memory (single-threaded C)

Illustration with a microbenchmark

Query: fill a histogram with jet p_T of all jets.

0.018 MHz	full framework (CMSSW, single-threaded C++)
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)

Illustration with a microbenchmark

Query: fill a histogram with jet p_T of all jets.

0.018 MHz	full framework (CMSSW, single-threaded C++)
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)

Illustration with a microbenchmark

Query: fill a histogram with jet p_T of all jets.

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)

Illustration with a microbenchmark

diana hep

Query: fill a histogram with jet p_T of all jets.

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)

Four orders of magnitude in performance lost to provide an object-oriented view of the jets, with all attributes filled.

Instead of turning TBranch data into objects, translate the code into loops over the raw TBranch arrays ("BulkIO").

User writes code with "event" and "jet" objects:

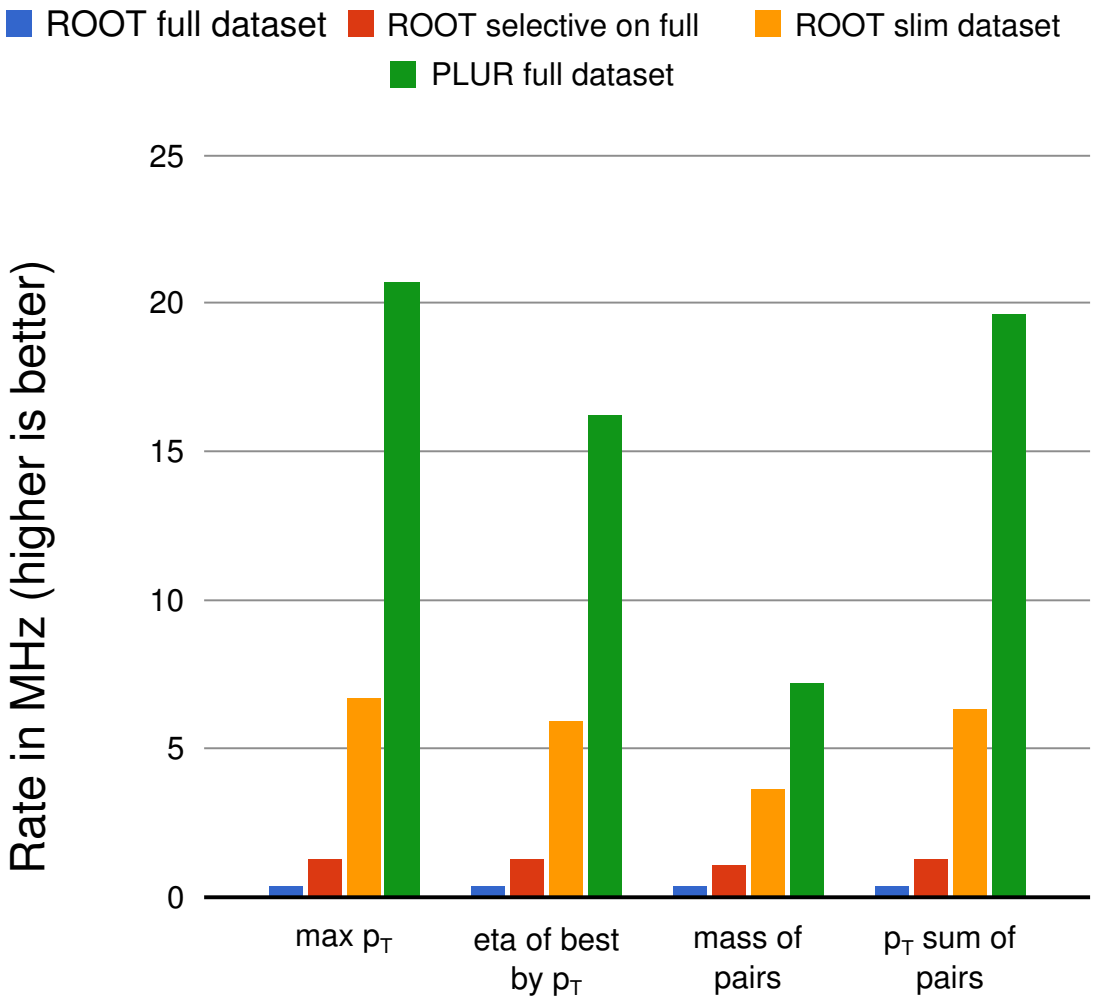
```
histogram = numpy.zeros(100, dtype=numpy.int32)

def fcn(roottree, histogram):
    for event in roottree:
        for jet in event.jets:
            bin = int(jet.pt)
            if bin >= 0 and bin < 100:
                histogram[bin] += 1
```

which are translated into indexes over raw arrays:

```
void fcn(int* events, int* jets, float* jetptdata, int* histogram) {
    int event, jet;
    for (event = 0;  event < events[1];  event++)
        for (jet = jets[event];  jet < jets[event + 1];  jet++) {
            int bin = (int)jetptdata[jet];
            if (bin >= 0  &&  bin < 100)
                histogram[bin] += 1;
}}
```

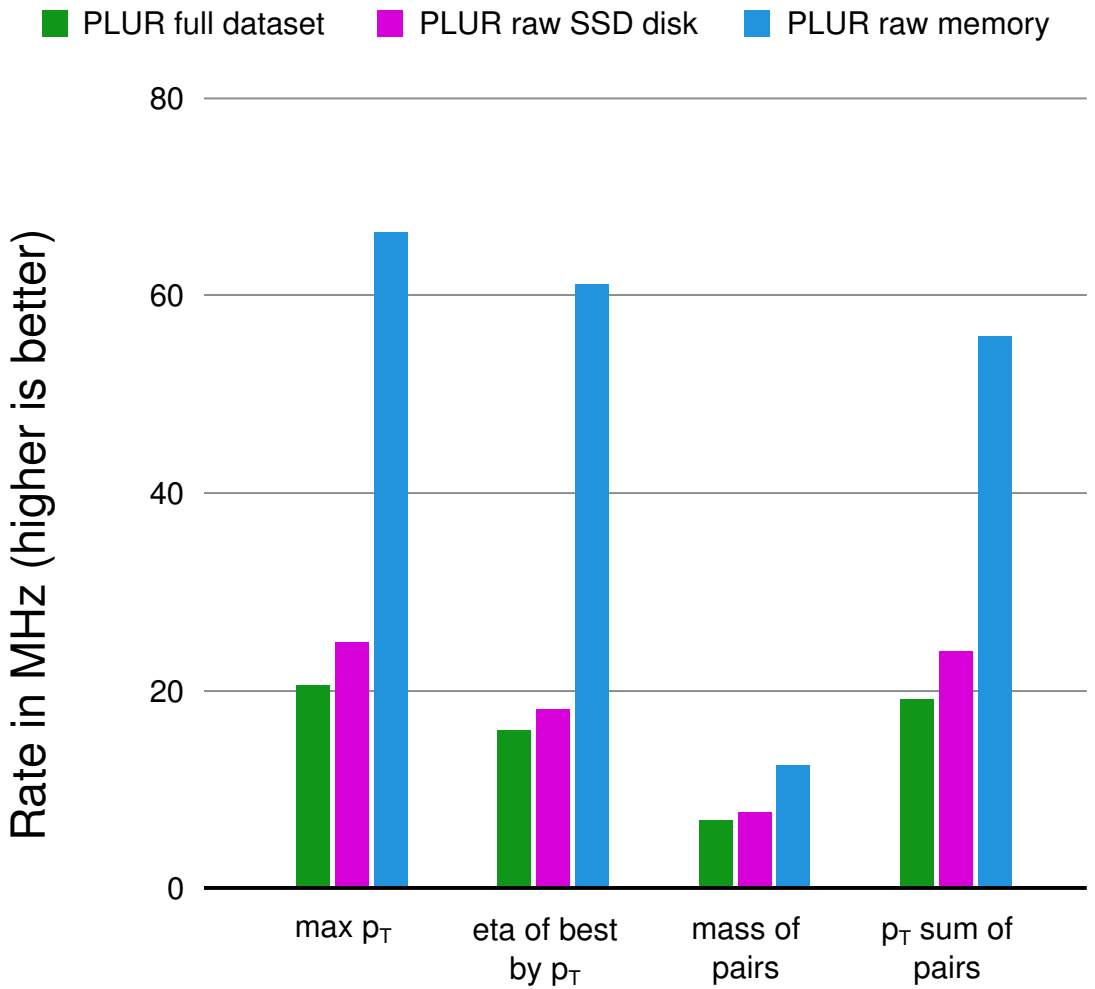
Performance on real queries

Four sample queries (see backup) using ROOT 1. without dropping branches, 2. dropping branches (`SetBranchStatus`), 3. using a slimmer file, 4. with BulkIO and transformed Python code.



Performance on real queries

Four sample queries (see backup) using 4. same ROOT with BulkIO and transformed Python code, 5. raw arrays on SSD disk, 6. raw arrays in memory. (Caching system for query server.)



You can try this out

Our PLUR library[1] puts **P**rimitive, **L**ist, **U**nion, and **R**ecord data into flat arrays and translates Python code to do array lookups.

The translated Python is then compiled by Numba[2] (Python compiler for numerical algorithms) into fast bytecode.

BulkIO contributions to ROOT[3] allow ROOT TBranch data to be rapidly viewed as Numpy arrays (10× faster than root_numpy).

This is being gathered into a HEPQuery application[4] to provide a query service. See the READMEs for more realistic examples.

---

[1] https://github.com/diana-hep/plur

[2] http://numba.pydata.org/

[3] https://github.com/bbockelm/root/tree/root-bulkapi-fastread-v2

[4] https://github.com/diana-hep/hepquery/

Distributed processing

(10's of MHz rates on previous page were all single-threaded)

Challenges: not a standard batch job

Overhead latency should be small enough for interactive use (less than a second).

Challenges: not a standard batch job

Overhead latency should be small enough for interactive use (less than a second).

Query-to-plot requires two phases: splitting into subtasks (map) and combining partial results (reduce). These must be coordinated.

Challenges: not a standard batch job

Overhead latency should be small enough for interactive use (less than a second).

Query-to-plot requires two phases: splitting into subtasks (map) and combining partial results (reduce). These must be coordinated.

Input data should be cached with column-granularity.

Challenges: not a standard batch job

Overhead latency should be small enough for interactive use (less than a second).

Query-to-plot requires two phases: splitting into subtasks (map) and combining partial results (reduce). These must be coordinated.

Input data should be cached with column-granularity.

Subtasks should *preferentially* be sent where their input data are cached (for data locality), but not *exclusively* (to elastically scale with dataset popularity).

Distributed state

To track ongoing jobs, accumulate partial histograms, and know where to find cached inputs, the query server will need to have distributed, mutable state.

Distributed state

To track ongoing jobs, accumulate partial histograms, and know where to find cached inputs, the query server will need to have distributed, mutable state.

Use third-party tools!

- Apache Zookeeper for rapidly changing task assignment.
- MongoDB for JSON-structured partial aggregations.
- Object store (Ceph?) for user-generated columns?
- . . . ?

Distributed state

To track ongoing jobs, accumulate partial histograms, and know where to find cached inputs, the query server will need to have distributed, mutable state.

Use third-party tools!

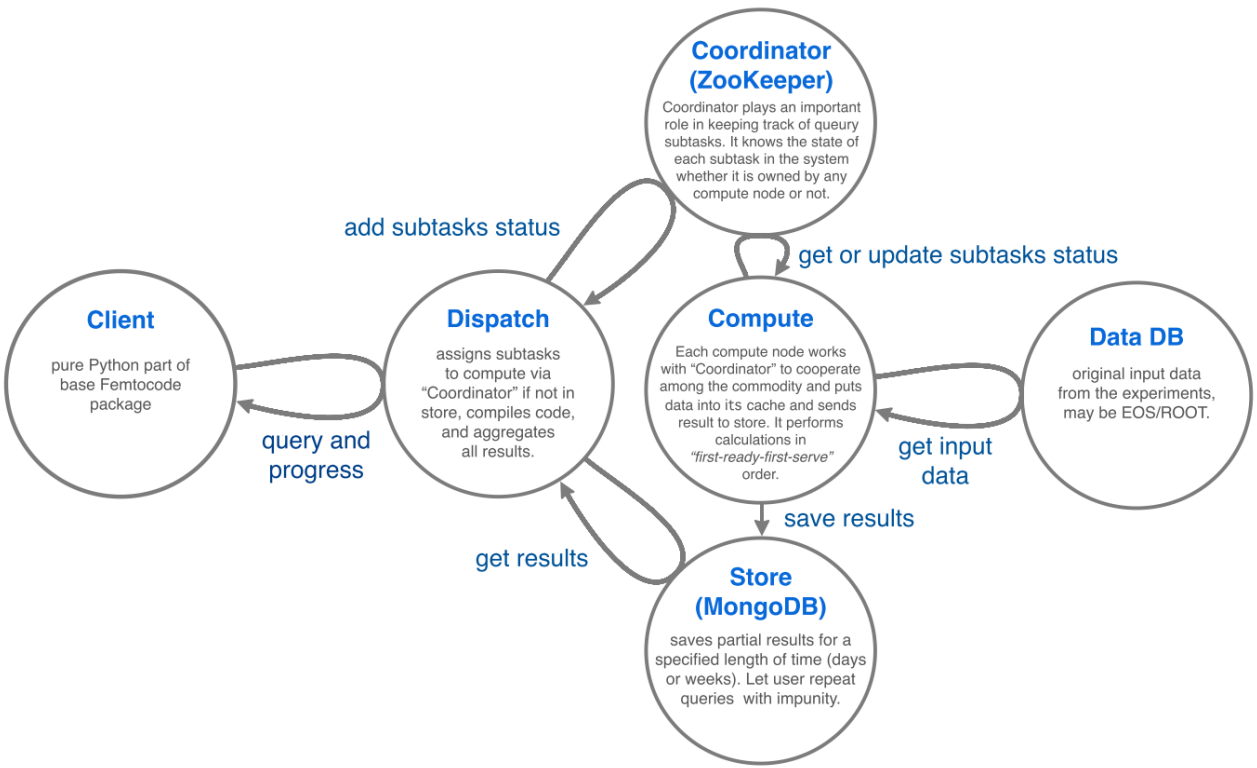
Apache Zookeeper

mongoDB.

- Apache Zookeeper for rapidly changing task assignment.
- MongoDB for JSON-structured partial aggregations.
- Object store (Ceph?) for user-generated columns?
- . . . ?

Although we can't find exactly what we want on the open-source market, we're finding most of the pieces.

Thanat Jatuphattharachat (CERN summer student) project: explore third party tools[1] and build a prototype system[2].



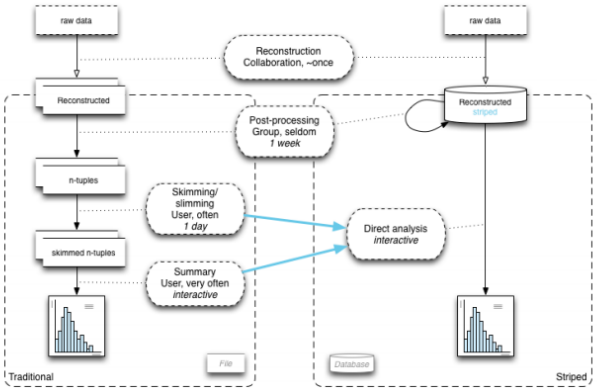
[1] https://cds.cern.ch/record/2278211

[2] https://github.com/JThanat/femto-mesos/tree/master

Striped Data Server for Scalable Parallel Data Analysis

Jin Chang (Caltech), Oliver Gutsche(FNAL), Igor Mandrichenko (FNAL), James Pivarski (Princeton University)

Goal: Reduce Time to Insight



Reduce analysis turn-around time

Programmatically, traditional analysis is iterative process, repeating:

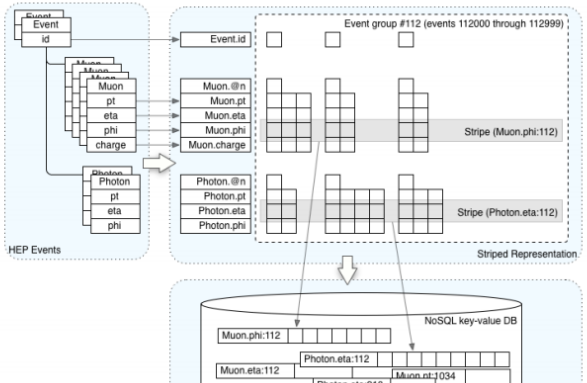
- Skimming (drop not interesting events, disk-to-disk)
- Slimming (drop unneeded attributes, disk-to-disk)
- Filtering (selectively read events into the memory)
- Pruning (selectively read attributes into the memory)

Provide access to the needed data and only to the needed data

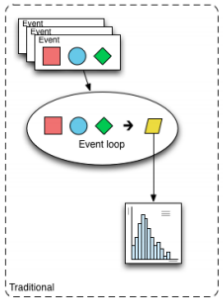
- Direct, scalable, efficient

Eliminate the need to skim or slim data as a disk-to-disk operation

Innovation: Striped sata representation – move from file-based to database-based analysis paradigm



Stripe is a numpy data array, immediately consumable by numpy



Traditional analysis:

Calculate missing quantities for each event, one event at a time in the event loop

Query language

Femtocode[1] is a mini-language based on Python, but with sufficient limitations to allow SQL-like query planning.

- Dependent type-checking to ensure that every query completes without runtime errors.
- Automated vectorization/GPU translation by controlling the loop structure.
- Integrates database-style indexing with processing.

But these are all "2.0" features.

[1] https://github.com/diana-hep/femtocode/

Conclusions

Conclusions!

- AOD-to-plot in seconds *is possible.*
- They're doing it in industry (but... SQL and Java).
- Python-based queries can be computed at single-threaded rates of 10–100 MHz by translating code, rather than deserializing data.
- Columnar data granularity has useful consequences.

  See Igor's poster!
- Prototyping distributed architecture, relying on third-party components wherever possible.

Backup

Sample queries

max p_T in Python

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

max p_T in C++

```
float maximum = 0.0;
for (i=0; i < muons.size(); i++)
  if (muons[i]->pt > maximum)
    maximum = muons[i]->pt;
fill_histogram(maximum);
```

eta of best by p_T in Python

```
maximum = 0.0
best = -1
for muon in event.muons:
  if muon.pt > maximum:
    maximum = muon.pt
    best = muon
if best != -1:
  fill_histogram(best.eta)
```

eta of best by p_T in C++

```
float maximum = 0.0;
Muon* best = nullptr;
for (i=0; i < muons.size(); i++)
  if (muons[i]->pt > maximum) {
    maximum = muons[i]->pt;
    best = muon; }
if (best != nullptr)
  fill_histogram(best->eta);
```

Sample queries

mass of pairs in Python

```
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)
```

mass of pairs in C++

```
int n = muons.size();
for (i=0; i < n; i++)
  for (j=i+1; j < n; j++) {
    Muon* m1 = muons[i];
    Muon* m2 = muons[j];
    double mass = sqrt(
      2*m1->pt*m2->pt*(
      cosh(m1->eta - m2->eta) -
      cos(m1->phi - m2->phi)));
    fill_histogram(mass); }
```

p_T sum of pairs in Python

```
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)
```

p_T sum of pairs in C++

```
int n = muons.size();
for (i=0; i < n; i++)
  for (j=i+1; j < n; j++) {
    Muon* m1 = muons[i];
    Muon* m2 = muons[j];
    double s = m1->pt + m2->pt;
    fill_histogram(s); }
```