Femtocode: querying HEP data

Jim Pivarski

Princeton University – DIANA

April 17, 2017

Reminder of motivation

(I last talked about this on December 12.)

Query systems

In some industries, it is commonplace to query petabytes of data in real time, usually with SQL. (Ibis, Impala, Kudu, Drill, etc.)

Reminder of motivation

(I last talked about this on December 12.)

Query systems

In some industries, it is commonplace to query petabytes of data in real time, usually with SQL. (Ibis, Impala, Kudu, Drill, etc.)

For us, this would mean being able to do final analysis directly on collaboration-shared Analysis Object Datasets (AODs), without managing private skims.

Reminder of motivation

(I last talked about this on December 12.)

Query systems

In some industries, it is commonplace to query petabytes of data in real time, usually with SQL. (Ibis, Impala, Kudu, Drill, etc.)

For us, this would mean being able to do final analysis directly on collaboration-shared Analysis Object Datasets (AODs), without managing private skims.

However, these systems don't deal (well) with rich objects, like arbitrary-length lists of jets containing lists of tracks.

Reminder of motivation

(I last talked about this on December 12.)

Query systems

In some industries, it is commonplace to query petabytes of data in real time, usually with SQL. (Ibis, Impala, Kudu, Drill, etc.)

For us, this would mean being able to do final analysis directly on collaboration-shared Analysis Object Datasets (AODs), without managing private skims.

However, these systems don't deal (well) with rich objects, like arbitrary-length lists of jets containing lists of tracks.

Femtocode

I'm developing a query system whose performance permits real-time analysis, but is capable of complex manipulations, such as filtering tracks, picking pairs to compute invariant masses, etc.

Three interrelated parts

Language/compiler

- As familiar to the user as possible (objects, nested loops).
- But constrained to allow restructuring for fast execution (map/filter/reduce instead of for loops, total-functional).
- Extra-strength type system to eliminate runtime errors.

Execution engine

- Operate on contiguous columns of data (like "TLeaf"), not objects. Restructuring becomes changes in arrays of integers.
- No memory allocation at runtime, vectorizable loops.
- JIT-compiled. CPU target for now, but GPU is possible.

Distributed server

- Vending machine: queries go in, histograms (etc.) come out.
- Referential transparency eliminates the need for "sessions."

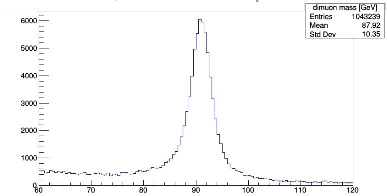
```
pending = session.source("ZZ_13TeV_pythia8")
    .define(mumass = "0.105658")      # chain of operations on source
    .toPython(mass = """
muons.map(mu1 => muons.map({mu2 =>    # doubly nested loop over muons
  p1x = mu1.pt * cos(mu1.phi);
  p1y = mu1.pt * sin(mu1.phi);        # shares scope with other steps
  p1z = mu1.pt * sinh(mu1.eta);       # in the chain (see "mumass")
  E1 = sqrt(p1x**2 + p1y**2 + p1z**2 + mumass**2);

  p2x = mu2.pt * cos(mu2.phi);
  p2y = mu2.pt * sin(mu2.phi);
  p2z = mu2.pt * sinh(mu2.eta);
  E2 = sqrt(p2x**2 + p2y**2 + p2z**2 + mumass**2);

  px = p1x + p2x; py = p1y + p2y;
  pz = p1z + p2z; E = E1 + E2;

  # "if" is required to avoid sqrt(-x)
  if E**2 - px**2 - py**2 - pz**2 >= 0:
    sqrt(E**2 - px**2 - py**2 - pz**2)
  else:
    None   # output type is nullable
}))
""").submit()                         # asynchronous submission to
final = pending.await()               # watch result accumulate
```

Yes, we see the Z peak.



Taking this example apart (1/3)

- Femtocode always appears in quotes (like SQL). It is a big-data aggregation step in the midst of a traditional analysis.
- A query is a "workflow" from source to aggregation, compiled and submitted as one unit.

  e.g. `source("dataset").define(X).define(Y).histogrammar(Z)`
- Most Femtocode expressions are tiny (hence "femto"), scattered throughout a Histogrammar aggregation:

```
session.source("dataset")
    .define(goodmuons = """...""")   # define good muons
    .filter("goodmuons.size >= 2")   # cut on them
    .define(dimuon = """..."""       # define dimuons
    .bundle(                         # plot their attributes
        mass = bin(120, 0, 12, "dimuon.mass"),
        pt = bin(100, 0, 100, "dimuon.pt"),
        eta = bin(100, -5, 5, "dimuon.eta"),
        phi = bin(314, 0, 2*pi, "dimuon.phi + pi"),
                                     # also plot the muons
        muons = explode("goodmuons", "mu", bundle(
            pt = bin(100, 0, 100, "mu.pt"),
            eta = bin(100, -5, 5, "mu.eta"),
            phi = bin(314, -pi, pi, "mu.phi"))))
```

Taking this example apart (2/3)

- Doubly nested loop by nesting functionals:
  `muons.map(mu1 => muons.map(mu2 => f(mu1, mu2)))`
  is equivalent to

  ```
  list_of_lists = []
  for mu1 in muons:
      list_of_numbers = []
      for mu2 in muons:
          list_of_numbers.append(f(mu1, mu2))
      list_of_lists.append(list_of_numbers)
  ```

- There will someday be more convenient forms: `pairs`, `table`, `filter`, `flatten`, `flatMap`, `zip`, `permutations`, etc.

  (This example would ideally use `pairs`, to avoid double-counting, and `flatten` to destructure the list-of-lists.)

Taking this example apart (3/3)

- Type system requires domain of `sqrt` to be guarded:

```
sqrt(E**2 - px**2 - py**2 - pz**2)
```

```
FemtocodeError: Function "sqrt" does not accept arguments with
the given types:

    sqrt(real)

    The sqrt function can only be used on non-negative numbers.

Check line:col 19:2 (pos 401):
      sqrt(E**2 - px**2 - py**2 - pz**2)
------^
```

But this works:

```
if E**2 - px**2 - py**2 - pz**2 >= 0:
    sqrt(E**2 - px**2 - py**2 - pz**2)
else:
    None
```

- The compiler tracks each subexpression's interval of validity:

`E**2 - px**2 - py**2 - pz**2` is limited to `real(min=0, max=inf)`.

In the future, we could use SymPy to discover this algebraically.

Another thing to notice

```
muons.map(mu1 => muons.map({mu2 =>
  p1x = mu1.pt * cos(mu1.phi);
  p1y = mu1.pt * sin(mu1.phi);
  p1z = mu1.pt * sinh(mu1.eta);
  E1 = sqrt(p1x**2 + p1y**2 + p1z**2 + mumass**2);
```

only uses `mu1`

```
  p2x = mu2.pt * cos(mu2.phi);
  p2y = mu2.pt * sin(mu2.phi);
  p2z = mu2.pt * sinh(mu2.eta);
  E2 = sqrt(p2x**2 + p2y**2 + p2z**2 + mumass**2);
```

only uses `mu2`

```
  px = p1x + p2x;
  py = p1y + p2y;
  pz = p1z + p2z;
  E = E1 + E2;

  if E**2 - px**2 - py**2 - pz**2 >= 0:
    sqrt(E**2 - px**2 - py**2 - pz**2)
  else:
    None
}))
```

uses both.

Femtocode minimizes computation

In most compilers, at least one of the two stanzas would be needlessly recomputed for every *pair* of muons. Physicists have learned to move these expressions out of the loop, possibly at the expense of readability.

Femtocode's compiler turns every loop over objects into vectorized functions on individual fields. A by-product of this is that the functions depending on `mu1` and `mu2` decouple from the functions depending on `(mu1, mu2)`.

In fact, all duplicate expressions are computed exactly once. The *only* reason to use assignment is for clarity.

What the dimuon example expands to

```
Sized by muons[]@size:
    #0       := cos(muons[]-phi)
    #1       := *(muons[]-pt, #0)
    #2       := **(#1, 2)
    #3       := sin(muons[]-phi)
    #4       := *(muons[]-pt, #3)
    #5       := **(#4, 2)
    #6       := sinh(muons[]-eta)
    #7       := *(muons[]-pt, #6)
    #8       := **(#7, 2)
    #9       := +(#2, #5, #8, 0.011164)
    #10      := sqrt(#9)
type(#10) == real(0.105658, almost(inf))

Sized by #11@size:
    #11@size := $explodesize(muons[], muons[])
    #11      := $explodedata(#10, #11@size, (muons[]))
    #12      := $explodedata(#10, #11@size, (muons[], muons[]))
    #13      := +(#11, #12)
    #14      := **(#13, 2)
    #15      := $explodedata(#1, #11@size, (muons[]))
    #16      := $explodedata(#1, #11@size, (muons[], muons[]))
    #17      := +(#15, #16)
    #18      := **(#17, 2)
    #19      := -(#14, #18)
    #20      := $explodedata(#4, #11@size, (muons[]))
    #21      := $explodedata(#4, #11@size, (muons[], muons[]))
    #22      := +(#20, #21)
    #23      := **(#22, 2)
    #24      := -(#19, #23)
    #25      := $explodedata(#7, #11@size, (muons[]))
    #26      := $explodedata(#7, #11@size, (muons[], muons[]))
    #27      := +(#25, #26)
    #28      := **(#27, 2)
    #29      := -(#24, #28)
    #30      := >=(#29, 0)
    #31      := <(#29, 0)
    #32      := -(#24, #28)
    #33      := sqrt(#32)
    #34      := if(#30, #31, #33, None)
type(#34) == union(null, real(0, almost(inf)))
```

`muons[]-pt`, `muons[]-phi`, `muons[]-eta`, `muons[]@size`, and everything that starts with a # is (at least conceptually) a big array of values.

All functions except `$explode*` are ideally suited to GPU acceleration.