

Femtoquery: query system for HEP

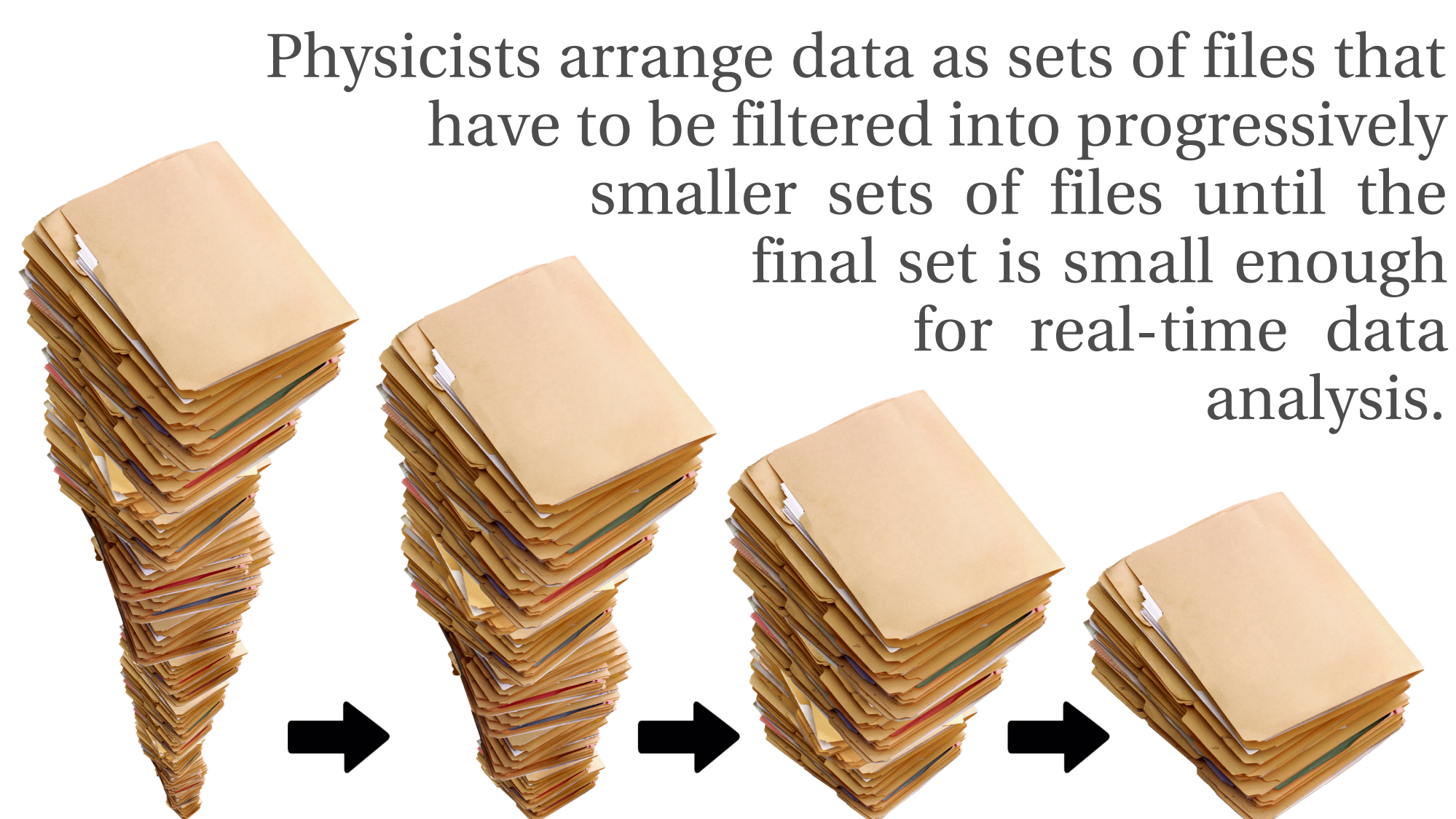
What's a query system?

User asks a question, gets an answer quickly enough to *explore* the data.

Like a Google query, but aggregating HEP data, returning (e.g.) histograms.

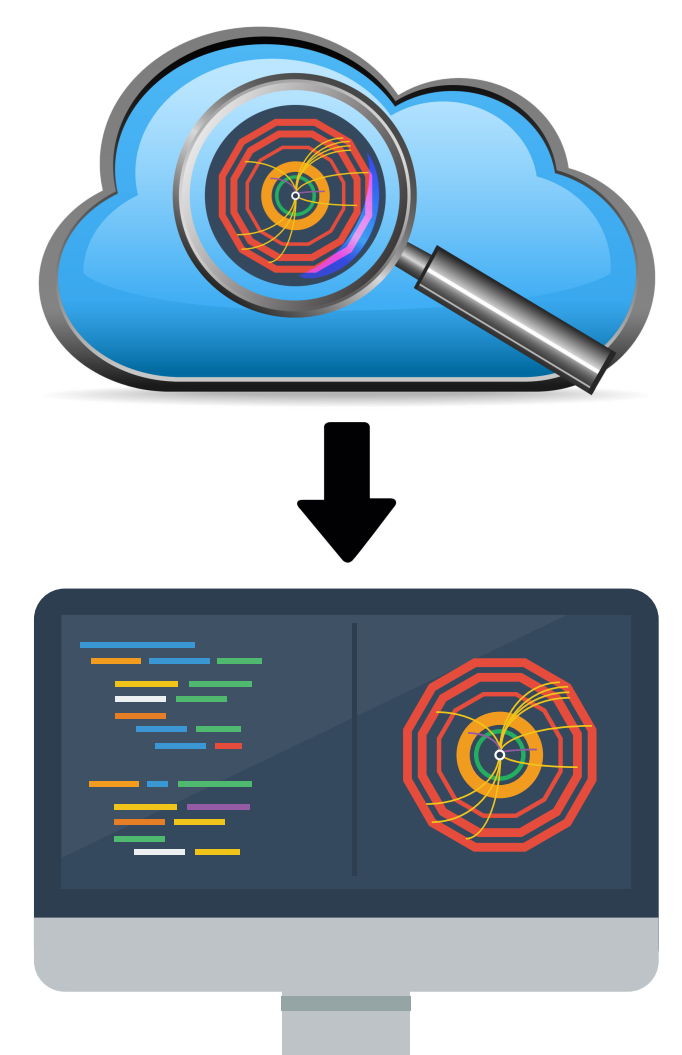
Embedded within an analysis script: provides sliced projections of the data for users to fit/plot/analyze in any way they want.

How it differs from what we do now



Instead, we propose a service that serves aggregated views of analysis object data on demand.

Must be responsive to requests in real-time: ~1 sec for each scan over a dataset.



High-level language

User writes expressions that pick apart the structure of objects within arbitrary-length lists, to any depth of nesting.

Higher-order functions like `.map`, `.pairs`, `.filter`, `.reduce` instead of explicit for loops.

Femtoquery query language is distributed in quoted snippets throughout a structured workflow and tree of aggregators.

(See <http://histogrammar.org> for histogram abstraction.)

```
workflow = session.source("b-physics")
.define(goodmuons = "muons.filter($1.pt > 5)") # pull from a named dataset
.filter("goodmuons.size >= 2") # muons with pt > 5 are good
.define(dimuon = """
    mu1, mu2 = goodmuons.maxby($1.pt, 2); # pick the top two by pt
    energy = mu1.E + mu2.E; # compute imploded energy/momentum
    px = mu1.px + mu2.px;
    py = mu1.py + mu2.py;
    pz = mu1.pz + mu2.pz;

    rec(mass = sqrt(energy**2 - px**2 - py**2 - pz**2),
        pt = sqrt(px**2 + py**2),
        phi = atan2(py, px),
        eta = ln((energy + pz)/(energy - pz))/2) # construct a record as output
    """)
.bundle( # make a bundle of plots
    mass = bin(120, 0, 12, "dimuon.mass"), # using the variables we've made
    pt = bin(100, 0, 100, "dimuon.pt"),
    eta = bin(100, -5, 5, "dimuon.eta"),
    phi = bin(314, 0, 2*pi, "dimuon.phi + pi"),
    muons = loop("goodmuons", "mu", bundle( # also make plots with one muon per entry
        pt = bin(100, 0, 100, "mu.pt"),
        eta = bin(100, -5, 5, "mu.eta"),
        phi = bin(314, -pi, pi, "mu.phi")
    ))
)

pending = workflow.submit() # submit the query
pending["mass"].plot() # and plot results while they accumulate
pending["muons"]["pt"].plot() # (they'll be animated)

blocking = pending.await() # stop the code until the result is in

massplot = blocking.plot.root("mass") # convert to a familiar format, like ROOT
massplot.Fit("gaus") # and use that package's tools
```

No runtime errors:

Type system includes min/max ranges of numbers and collection lengths, so that out-of-domain errors are caught at compile-time.

Examples: "x/y" must be "if y != 0: x/y else: None".

`filter("goodmuons.size >= 2")` above ensures that we can assign two of them as `mu1, mu2`.

Total functional language:

Every expression that compiles returns a result. (For simplicity, recursion is not allowed.)

Low-level execution

No objects at runtime:

All nested structures are represented as homogeneous arrays.

type = `collection(collection(record(a=integer, b=real)))`
values = `[[(1, 1.1)] [(2, 2.2)] (3, 3.3)] [(4, 4.4)]]`
becomes
`data[]@size = 3 1 0 2 3 3 1 1`
`data[]@-a = 1 2 3 4`
`data[]@-b = 1.1 2.2 3.3 4.4`

```
muons.map( {mu1 =>
    muons.map( {mu2 =>
        e1 = mu1.p**2 + 0.105658**2;
        e2 = mu2.p**2 + 0.105658**2;
        e1 + e2
    }).max
}).max
```

physicist unnecessarily wrote these lines in the loop over muon pairs

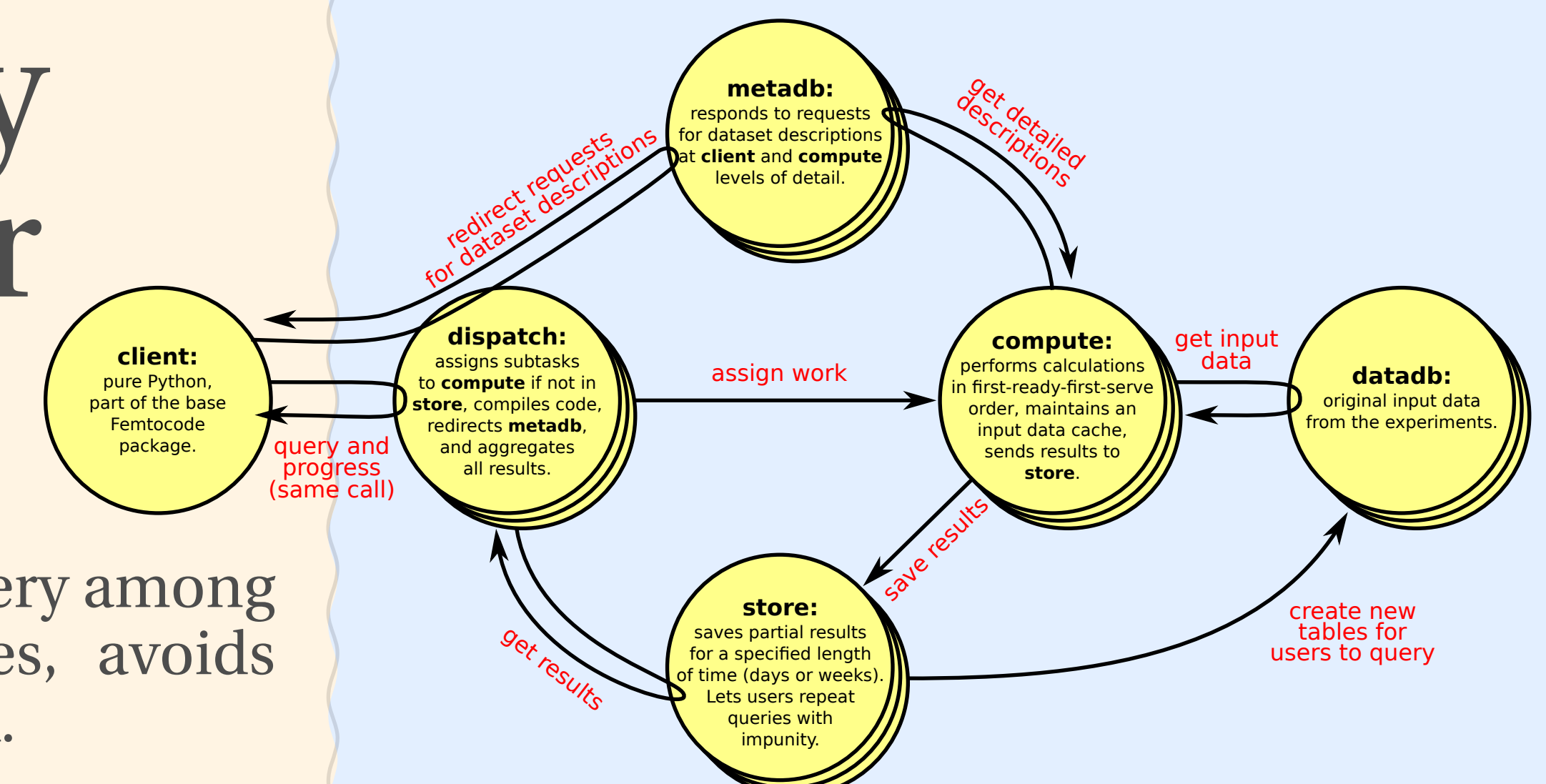
```
#0 := *(muons[]-p, 2)
#1 := +(#0, 0.011164)
#2@size := $explodesize(muons[], muons[])
#2 := $explodedata(#1, #2@size, (muons[]))
#3 := $explodedata(#1, #2@size, (muons[], muons[]))
#4 := +(#2, #3)
#5 := $implode(#4, muons[], "max")
#6 := $implode(#5, None, "max")
```

loop over muons[]@size over #2@size

Each non-\$ statement can be a branchless, loopless GPU kernel or all statements can be grouped by @size as CPU loops.

Query server

Distributes query among compute nodes, avoids duplicate work.



Performance studies

- Typical analysis asks for ~10 TB of input data.
- Typical query touches < 1% of the *columns*.
- Disk read at 40 MB/sec ⇒ 2.6 sec for 1000 disks.
- Typical analysis touches < 10% of the *columns*.
- Therefore, fill 1 TB of a cluster's RAM as cache.
- Cache read at 1 GB/sec ⇒ 0.1 sec for 1000 cores.

Microbenchmarks to get a sense of scale:

Just adding a constant to an array of 64-bit floats on different systems. The first are disk limited, the last are memory bandwidth limited.

0.018 MHz	CMSW EDAAnalyzer (C++ event framework)
1.5 MHz	TTree::Draw in ROOT (HEP analysis toolkit)
2.8 MHz	minimal disk read and unzip
12 MHz	allocating C++ objects on heap, iterating, deleting
31 MHz	allocating C++ objects on stack and iterating
54 MHz	Femtoquery loop, current implementation
250 MHz	minimal single-threaded loop in C (our goal)
8 000 MHz	same loop on 128 threads in KNL's MCDRAM
57 000 MHz	equivalent on Tesla P100-SXM2 GPU

