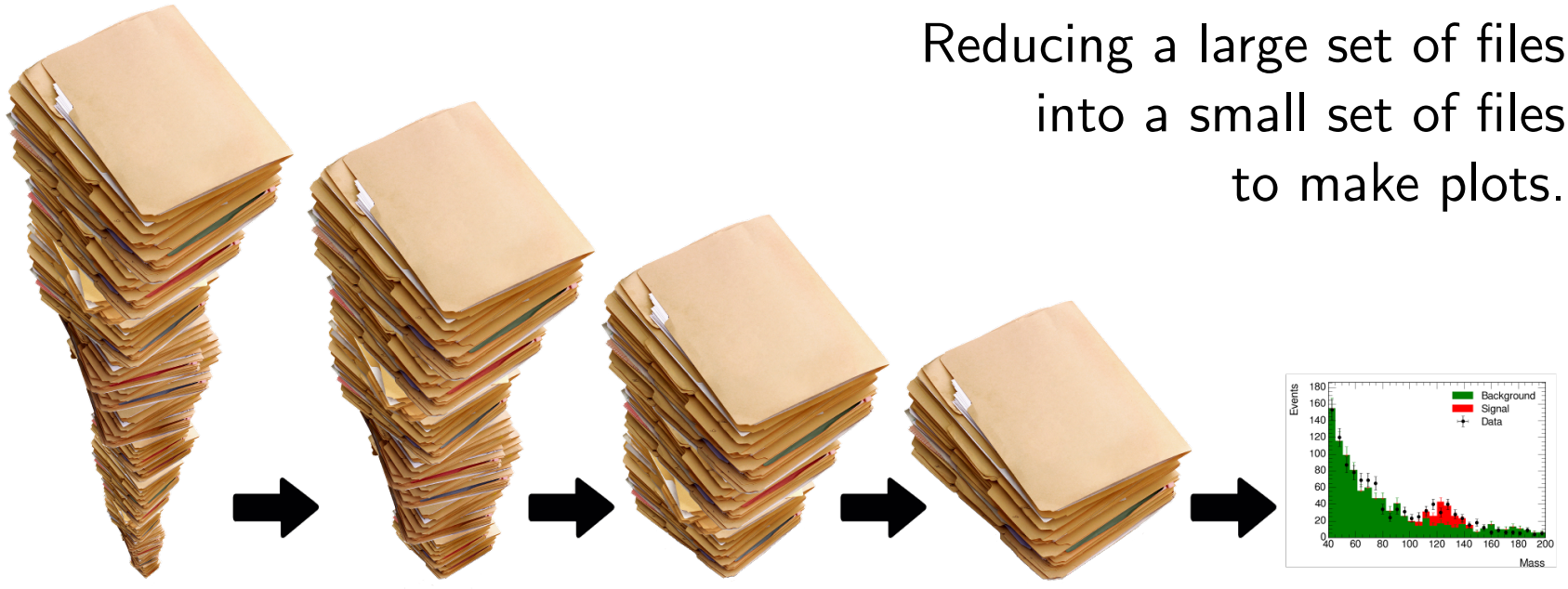
Femtocode: querying HEP data

Jim Pivarski

Princeton University – DIANA

May 23, 2017

A data analyst's life



Reducing a large set of files
into a small set of files
to make plots.

The last dataset must be small enough to permit real time plotting and re-plotting.

This is a problem

Assertion: if physicists *could* make plots (and other aggregations for statistical analysis) directly from the collaboration's Analysis Object Data in real time, they *would*.

Database-style interaction

Rapid queries on big data are possible; in fact, it's a big field:

Database-style interaction

Rapid queries on big data are possible; in fact, it's a big field:

- Instead of users maintaining private skims, running local processes, they share a distributed query server.

Database-style interaction

Rapid queries on big data are possible; in fact, it's a big field:

- Instead of users maintaining private skims, running local processes, they share a distributed query server.
- Instead of fetching data from disk, cache in RAM/SSD/X-Point.

Database-style interaction

Rapid queries on big data are possible; in fact, it's a big field:

- Instead of users maintaining private skims, running local processes, they share a distributed query server.
- Instead of fetching data from disk, cache in RAM/SSD/X-Point.
- Instead of evaluating user's code as-is, translate it into a form that optimizes memory bandwidth.

Database-style interaction

Rapid queries on big data are possible; in fact, it's a big field:

- Instead of users maintaining private skims, running local processes, they share a distributed query server.
- Instead of fetching data from disk, cache in RAM/SSD/X-Point.
- Instead of evaluating user's code as-is, translate it into a form that optimizes memory bandwidth.
- Instead of executing operations on nested objects, execute on flat arrays of numbers.

Sharing and caching go together

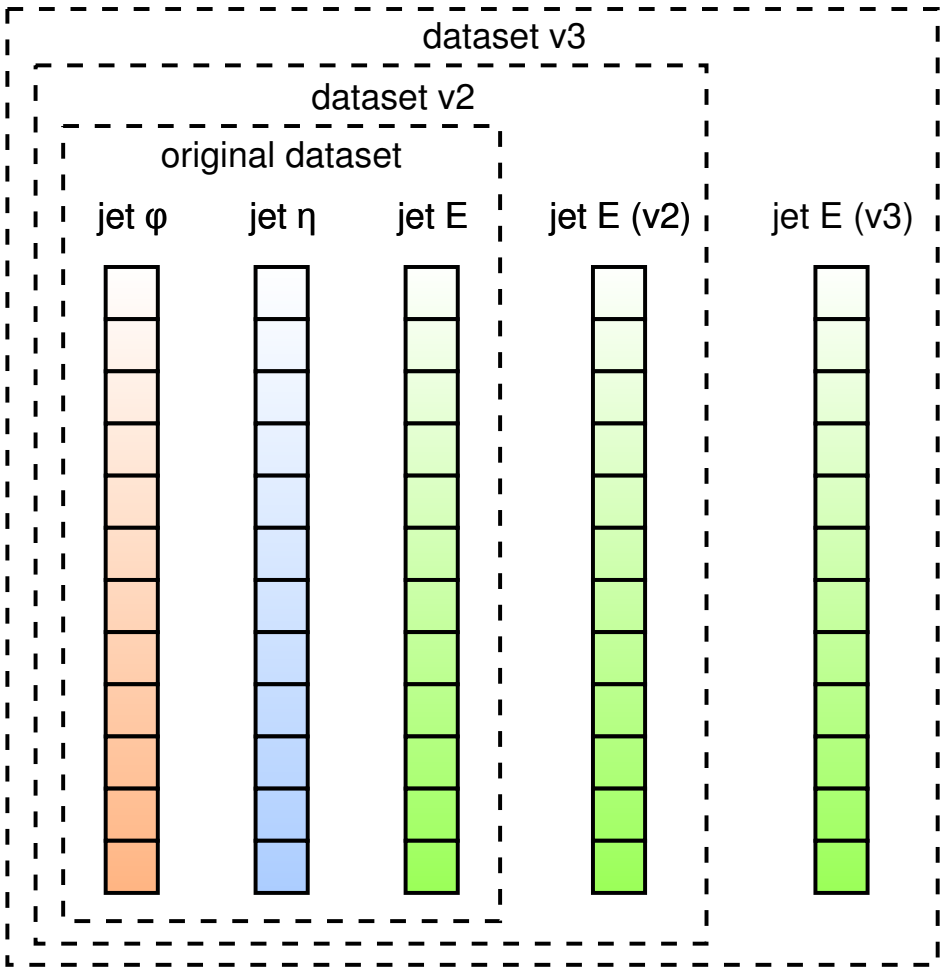
Most users need *mostly* the same input variables.

- For instance, all muon analyses use the same kinematic variables and might also use different isolation variables.
- Exact fraction is unknown, but we know that analysis groups share ntuplizers that slim the same 10% (4 kB/event) of CMS MiniAOD.

If, say, half of the variables are shared and half are not, a server distributed across a cluster with 10 TB of RAM effectively gives each user 5 TB + ϵ of RAM.

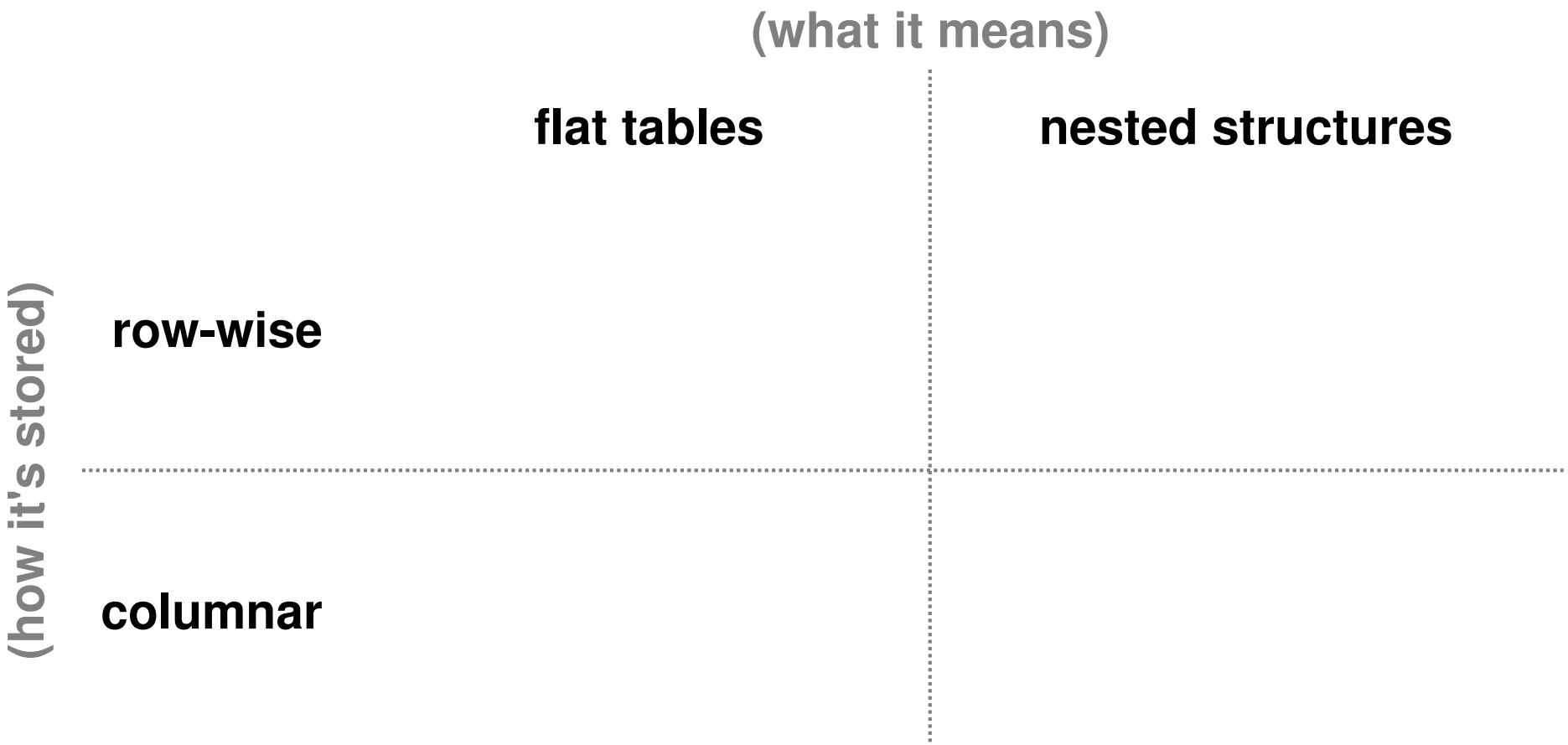
Sharing and columnar data go together

Columns for a dataset do not need to come from the same file.



Extreme version of the "friend TTree" concept (superfriends).

The limitation



The limitation

The limitation

(what it means)

(how it's stored)

	flat tables	nested structures
row-wise		
columnar		?

Femtocode: query language and engine

Physicist's view

Muon object schema:

```
collection(record(
    pt = real(0, almost(inf)),
    eta = real,
    phi = real(-pi, pi)))
```

Example query:

```
muons.filter(mu => mu.pt > 5)
     .map(mu => mu.pt*sinh(mu.eta))
     .max
```

Return type:

```
union(null, real)
```

Execution engine

Physical representation:

```
muons.pt:    [31.09, 9.76, 8.18, ...]
muons.phi:   [-0.48, 0.12, 0.12, ...]
muons.eta:   [0.88, 0.92, -0.26, ...]
muons@size: [3, 1, 1, 2, ...]
```

Example execution:

1. Compute `muons.pt > 5` for *all* muons.
2. Compute `sinh(muons.eta)` for elements in which `#1` is true.
3. Compute `muons.pt * #2` for elements in which `#1` is true.
4. Pick the maximum `#3` value or `NaN`, returning array with one value per event.

Representing objects as flat arrays

For simple collections of records (e.g. particles), these arrays have the same interpretation as ROOT TLeaves:

- data arrays contain all values, ignoring event boundaries,
- size array contains the size of each event's collection.

Representing objects as flat arrays

For simple collections of records (e.g. particles), these arrays have the same interpretation as ROOT TLeaves:

- data arrays contain all values, ignoring event boundaries,
- size array contains the size of each event's collection.

Except for runtime calculations, rather than just on disk.

Representing objects as flat arrays

For simple collections of records (e.g. particles), these arrays have the same interpretation as ROOT TLeaves:

- data arrays contain all values, ignoring event boundaries,
- size array contains the size of each event's collection.

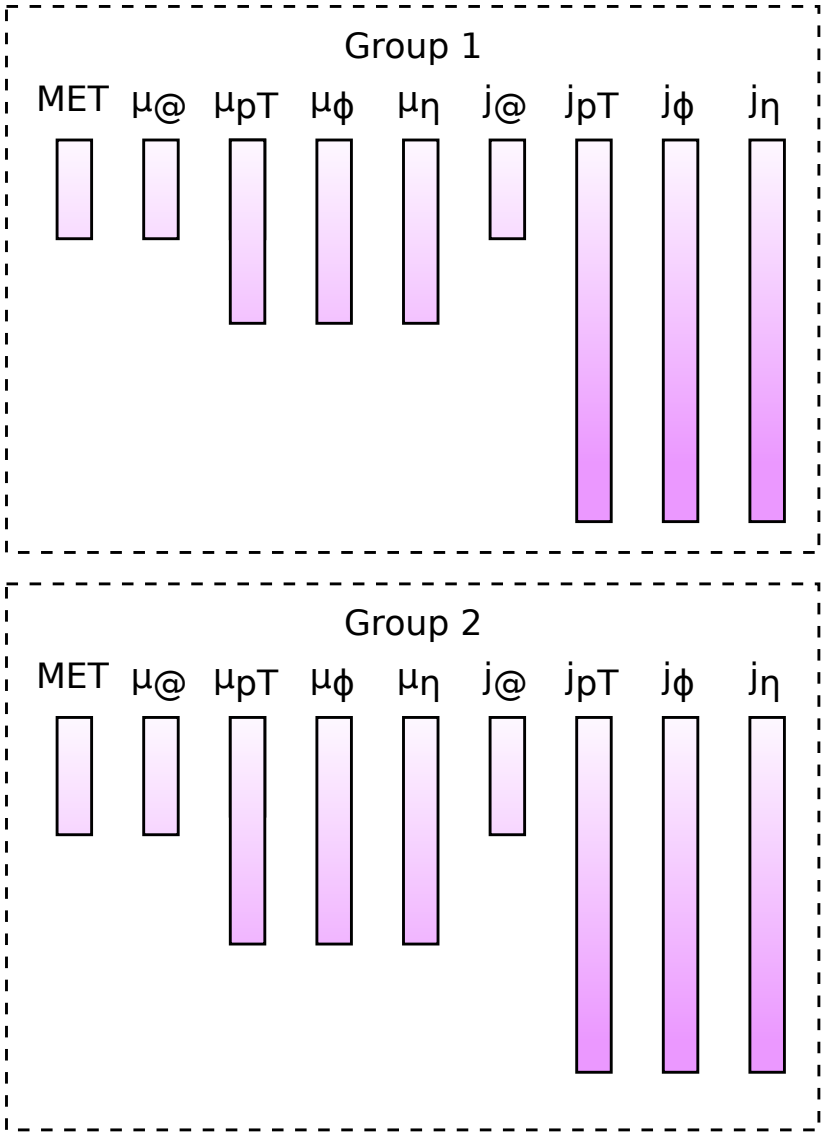
Except for runtime calculations, rather than just on disk.

For collections of collections (with fixed, known depth), we can extend this definition recursively:

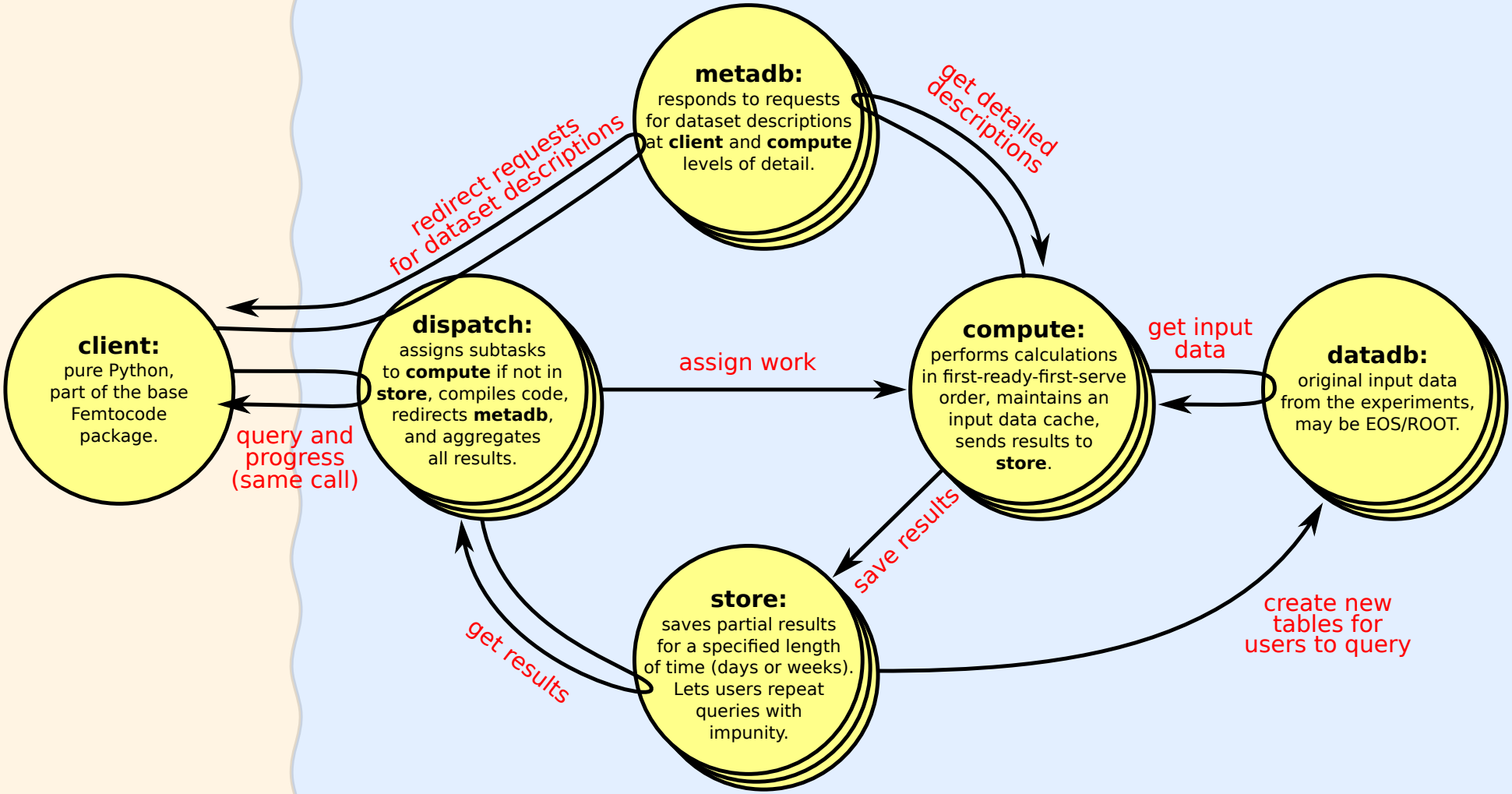
```
Given:             [ [ a b c ] [ d e f g ] ] [ [ h ] [ i j ] ]
Data array:          a b c     d e f g         h     i j
Recursive counter: 2 3         4             2 1     2
```

Granular unit of parallelization

Each query might touch any column, so the unit of parallelization is an group of columns with an integer number of events.



Query system layout



Working example: dimuons

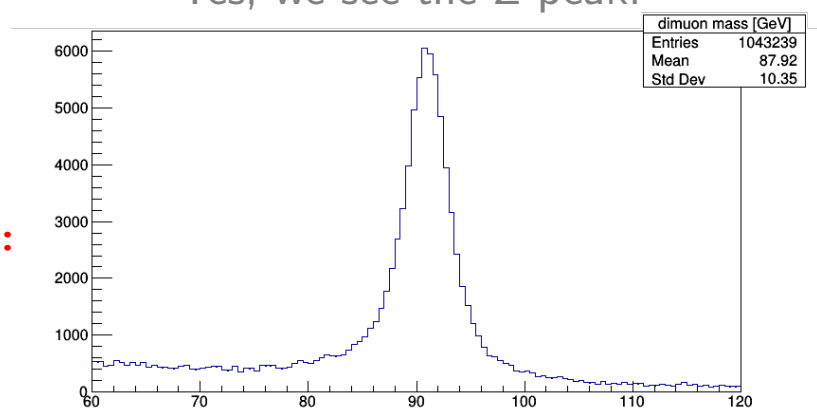
```
pending = session.source("ZZ_13TeV_pythia8")
    .define(mumass = "0.105658")      # chain of operations on source
    .toPython(mass = """
muons.map(mu1 => muons.map({mu2 =>    # doubly nested loop over muons
  p1x = mu1.pt * cos(mu1.phi);
  p1y = mu1.pt * sin(mu1.phi);        # shares scope with 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.



Comparing apples and bananas

I/O-bounded operation (plus) on 806 177 jet p_T values (6.15 MB):

0.018 MHz	CMSSW EDAnalyzer (too many branches loaded)
1.5 MHz	`TTree::Draw` into single-bin histogram
2.8 MHz	minimal disk read and unzip (ROOT or Numpy)
12 MHz	allocating C++ objects on heap, iterating, deleting
31 MHz	allocating C++ objects on stack, iterating
54 MHz	current implementation of Femtocode loop*
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

*Not final.

Concluding remarks

Shared query-based data access is worth pursuing.

Femtocode is an implementation of that idea,
and is starting to work.

Taking this example apart (1/3)

- Femtocode always appears in quotes (like SQL). It is a big-data aggregation step that feeds into 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 snippets 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 = loop("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)

- Loop over pairs of muons is constructed 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)

return list_of_lists
```

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

  (The dimuon example would ideally use `pairs` to avoid double-counting and `flatten` to destructure the list-of-lists. Or better yet, pick two by p_T to get one candidate per event.)

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)
------^
```

  To resolve this compile-time error, we write:

```
  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);

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

only uses `mu1`

only uses `mu2`

uses both.

The dimuon example, after "compilation"

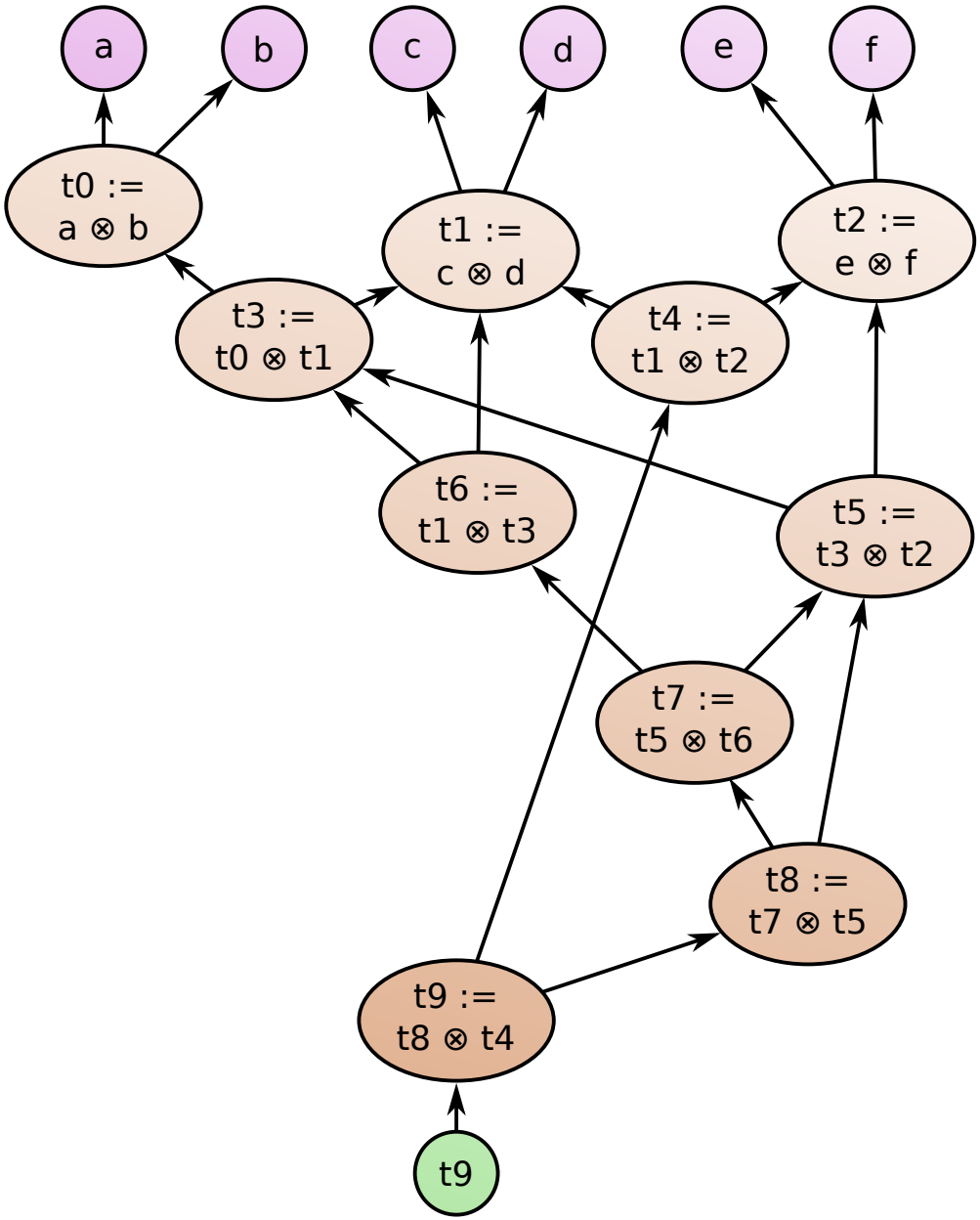
```
Sized by muons[]@size:
    #0        := cos(muons[]-phi)
    #1        := *(muons[]-pt, #0)
    #2        := **(#1, 2)
    #3        := sin(muons[]-phi)
    #4        := *(muons[]-pt, #3)
    #5        := **(#4, 2)
    #6        := sinh(muons[]-eta)
    #7        := *(muons[]-pt, #6)
    #8        := **(#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*` would make good GPU kernels.

Freedom to choose the looping structure



Suppose we have this dependency graph.

We are free to choose where to put the loops.

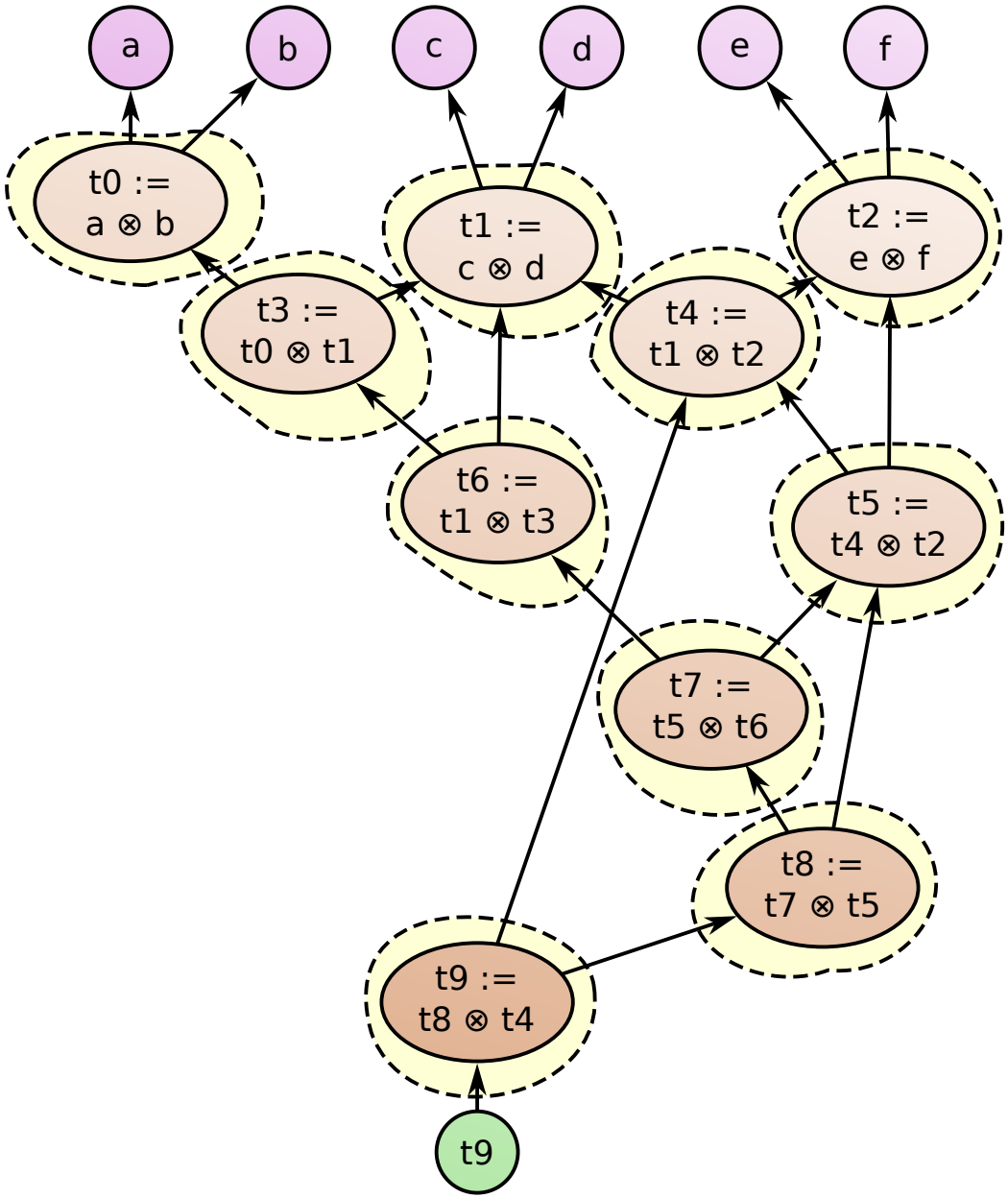
a, b, c, d, e, and f are all large arrays

t9 must also be a large array

intermediate steps need not be

(⊗ is some operation)

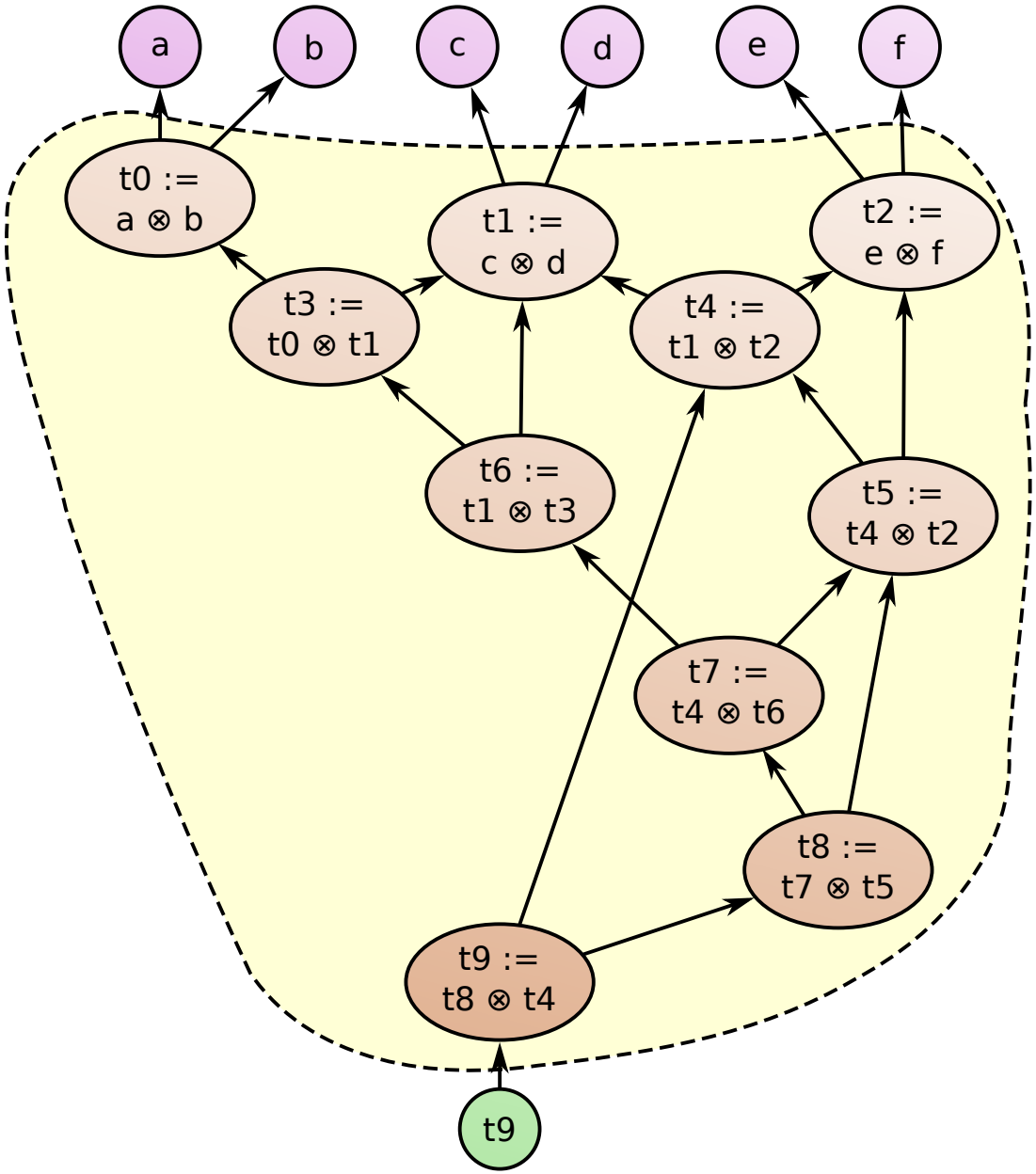
Freedom to choose the looping structure



At every step:

```
foreach i:
    t0[i] := a[i] ⊗ b[i]
foreach i:
    t1[i] := c[i] ⊗ d[i]
foreach i:
    t2[i] := e[i] ⊗ f[i]
foreach i:
    t3[i] := t0[i] ⊗ t1[i]
foreach i:
    t4[i] := t1[i] ⊗ t2[i]
foreach i:
    t5[i] := t4[i] ⊗ t2[i]
foreach i:
    t6[i] := t1[i] ⊗ t3[i]
foreach i:
    t7[i] := t5[i] ⊗ t6[i]
foreach i:
    t8[i] := t7[i] ⊗ t5[i]
foreach i:
    t9[i] := t8[i] ⊗ t4[i]
```

Freedom to choose the looping structure

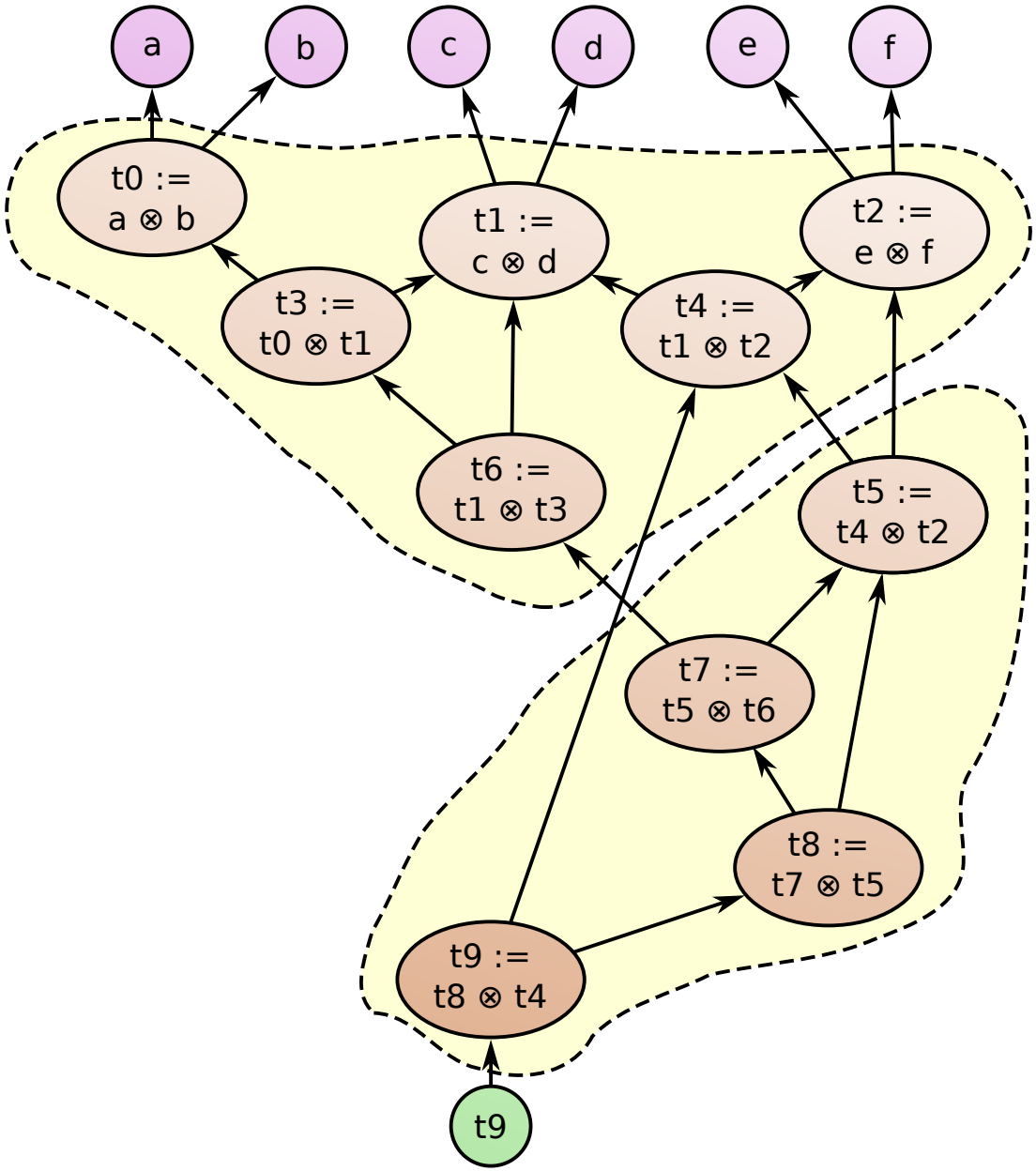


Around everything:

foreach i:

```
t0 := a[i] ⊗ b[i]
t1 := c[i] ⊗ d[i]
t2 := e[i] ⊗ f[i]
t3 := t0 ⊗ t1
t4 := t1 ⊗ t2
t5 := t4 ⊗ t2
t6 := t1 ⊗ t3
t7 := t5 ⊗ t6
t8 := t7 ⊗ t5
t9[i] := t8 ⊗ t4
```

Freedom to choose the looping structure



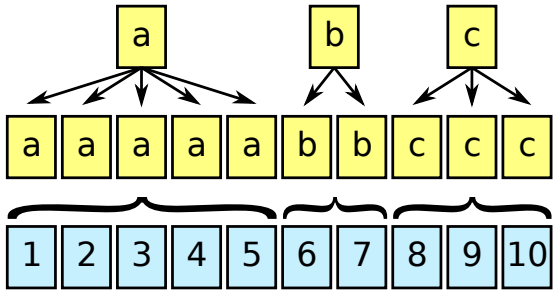
Or an intermediate case:

```
foreach i:
    t0 := a[i] ⊗ b[i]
    t1 := c[i] ⊗ d[i]
    t2[i] := e[i] ⊗ f[i]
    t3 := t0 ⊗ t1
    t4[i] := t1 ⊗ t2
    t6[i] := t1 ⊗ t3
foreach i:
    t5 := t4[i] ⊗ t2[i]
    t7 := t5 ⊗ t6
    t8 := t7 ⊗ t5
    t9[i] := t8 ⊗ t4
```

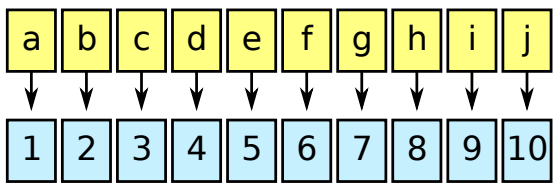
Note that this changes which quantities are arrays and which are scalars.

Three kinds of operations in each plateau

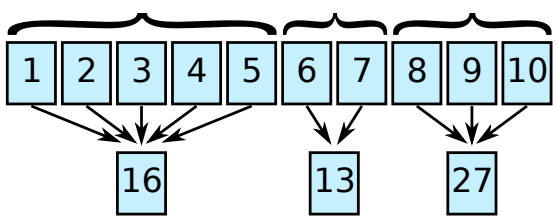
Explode: increase cardinality of one array so that it matches another. Determines the indexing of the loop, so must be first.



Flat: apply function to all members of two aligned data arrays, ignoring event boundaries. Intermediate steps need not be arrays.



Implode: combine results (sum, mean, max, etc.) to reduce cardinality of an array. Size of output arrays are not constrained by the indexing of the loop. Must be last.



Using ROOT functions in Femtocode

dianahep

```
########## ROOT/some_library.py, somewhere visible to nodes on the Femtocode server.
import ctypes
libMathCore = ctypes.cdll.LoadLibrary("libMathCore.so")
chi2_ctypes = libMathCore._ZN5TMath17ChisquareQuantileEdd    # c++filt!
chi2_ctypes.argtypes = (ctypes.c_double, ctypes.c_double)
chi2_ctypes.restype = ctypes.c_double
```

```
########## Creating a custom library (on the Femtocode client):
from femtocode.typesystem import *
from femtocode.lib.custom import *

def chi2_sig(x, n):
    # Compile-time type-safety: assert parameter types, provide return type.
    assert isinstance(x, Number) and \
        almost.min(0, x.min) == 0 and almost.max(x.max, almost(1)) == almost(1)
    assert isinstance(n, Number) and n.whole and n.min > 0
    return real(0, almost(inf))

custom = CustomLibrary()              # module name          symbol name    signature
custom.add(CustomFlatFunction("chi2", "ROOT.some_library", "chi2_ctypes", chi2_sig))

########## Running a Femtocode query that uses this library:
from femtocode.run.execution import NativeTestSession
session = NativeTestSession()

# Define a dataset with the right types and fill it with dummy data.
numerical = session.source("Test", x=real(0, almost(1)), n=integer(1, almost(inf)))
for i in range(100):
    numerical.dataset.fill({"x": i / 100.0, "n": i + 1})

# Femtocode calls TMath::ChisquareQuantile without involving Python at all.
result = numerical.toPython(out = "chi2(x, n)").submit(libs=[custom])
for entry in result:
    print entry
```

Dividing the problem

Femtocode's design philosophy is to do work up-front to streamline the event loop. In the distributed server, managing subtasks is part of this up-front work. Time to completion could be summarized as

$$\text{time} = C_1 + C_2(n_{\text{cores}}) \cdot N_{\text{subtasks}} + \frac{C_3}{n_{\text{cores}}} \cdot N_{\text{events}}$$

- C_1 is a constant, dominated by 70 ms of JIT-compilation time,
- $C_2(n_{\text{cores}})$ is the time spent managing subtasks, a complex concurrent processes affected by Amdahl's law.
- C_3 is the part that actually executes the user's query; it is natively compiled and embarrassingly parallel.

The order parameter in this problem is N_{events}. We get to choose $N_{\text{subtasks}}/N_{\text{events}}$ and can simply make partitions larger if the Pythonic "data management" part becomes an issue.