Femtocode: querying HEP data

Jim Pivarski

Princeton University – DIANA

April 17, 2017

Reminder of motivation

(The last time I presented this here was 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

(The last time I presented this here was 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.

(The last time I presented this here was 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 tracks containing hits. . .

Reminder of motivation

(The last time I presented this here was 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 tracks containing hits. . .

Femtocode

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

Three interrelated parts

diana hep

Language/compiler

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

Execution engine

- Operate on contiguous columns of data, not objects. "Restructuring objects" becomes changing arrays of integers.
- No memory allocation at runtime; vectorizable loops.
- JIT-compiled. CPU for now, but structure is right for GPU.

Distributed server

- Vending machine: queries go in, histograms (etc.) come out.
- Referential transparency eliminates the need of tracking users.

Langauge/Compiler

Start with a working example: dimuons

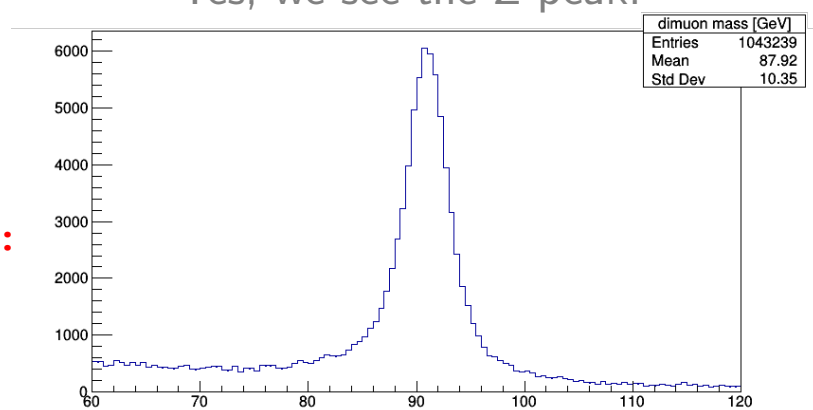
```
pending = session.source("ZZ_13TeV_pythia8")
    .define(mumass = "0.105658")      # chain of operations on source
    .toPython(mass = """
muons.map(mu1 => muons.map({mu2 =>    # doubly nested loop over muons
  p1x = mu1.pt * cos(mu1.phi);
  p1y = mu1.pt * sin(mu1.phi);        # shares scope with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



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.

Femtocode minimizes computation

In most compilers, at least one of those 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 just `mu1` or `mu2` decouple from the functions depending on both.

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

(It's like an executable whiteboard.)

Execution engine

The dimuon example, after "compilation"

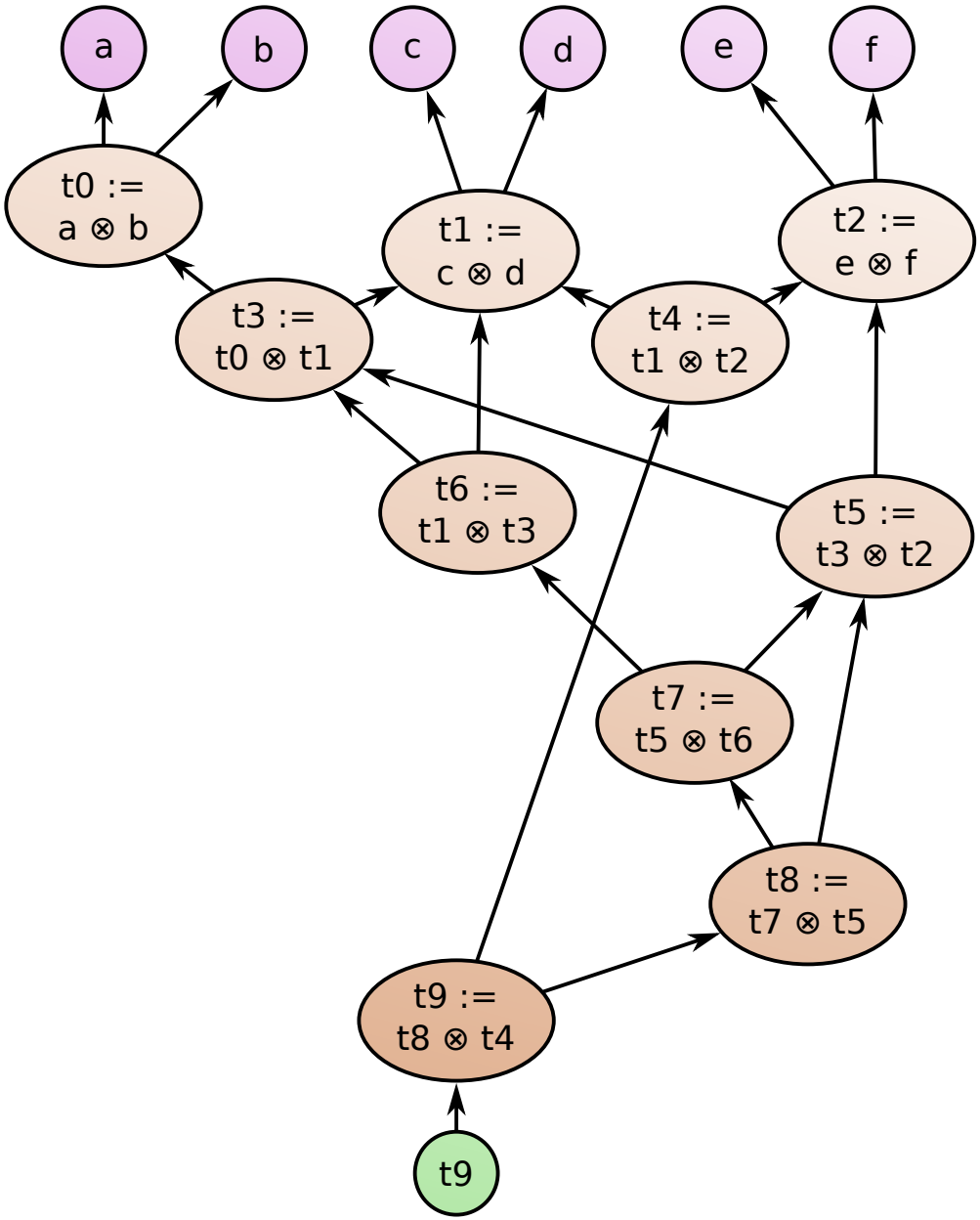
```
Sized by muons[]@size:
    #0        := cos(muons[]-phi)
    #1        := *(muons[]-pt, #0)
    #2        := **(#1, 2)
    #3        := sin(muons[]-phi)
    #4        := *(muons[]-pt, #3)
    #5        := **(#4, 2)
    #6        := sinh(muons[]-eta)
    #7        := *(muons[]-pt, #6)
    #8     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

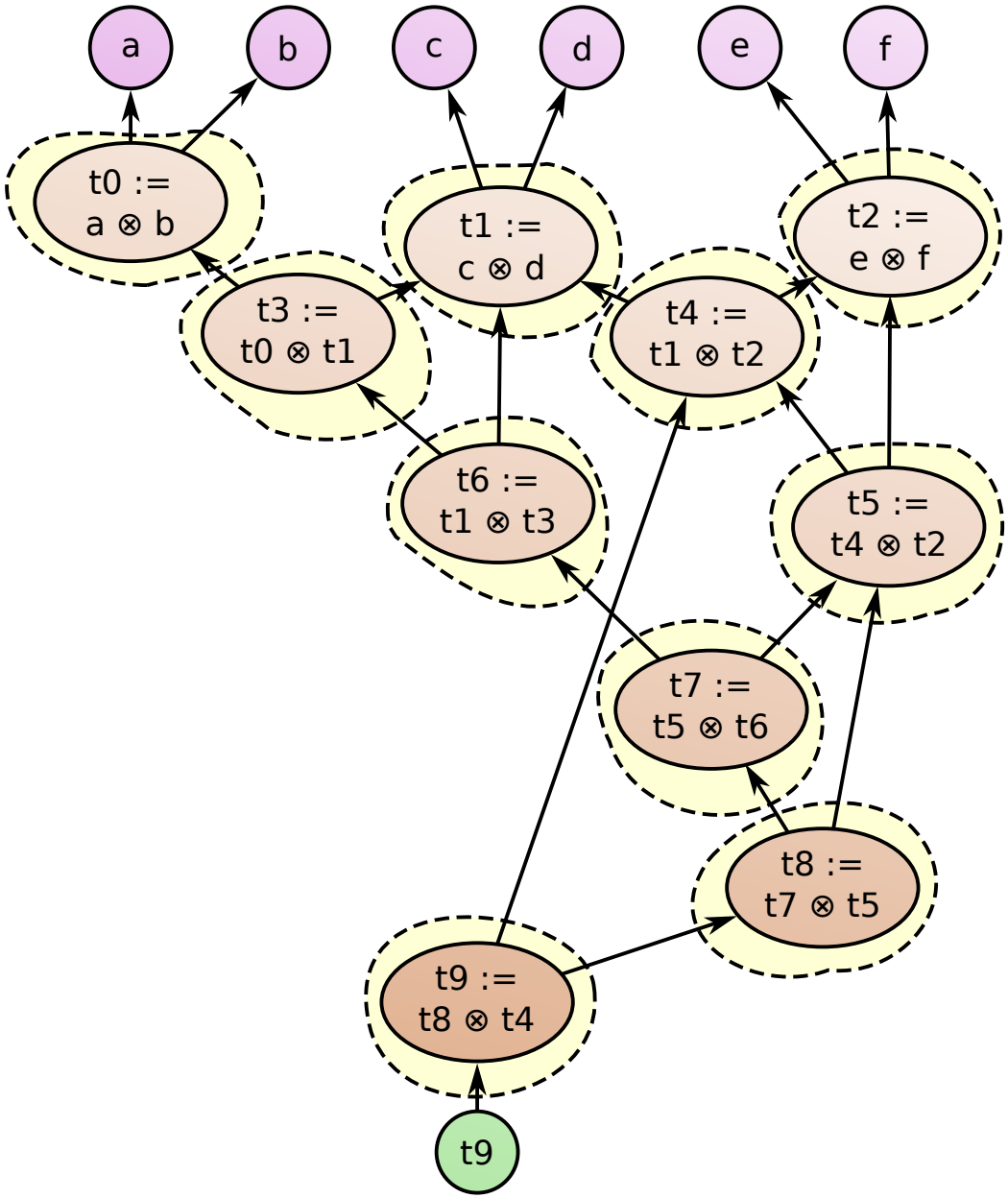
a, b, c, d, e, and f are all large arrays

t9 must also be a large array

intermediate steps need not be

(⊗ is some operation)

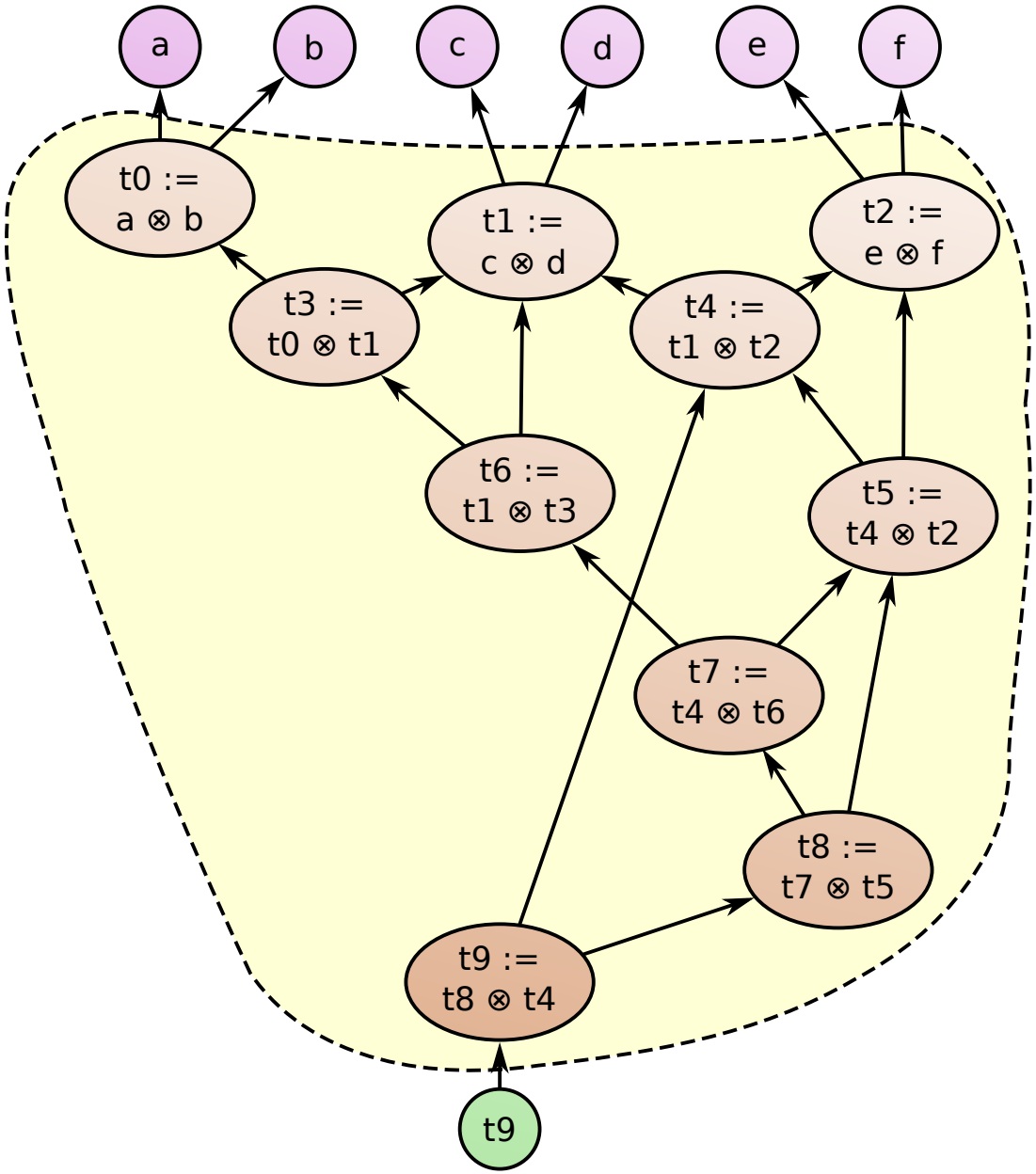
Freedom to choose the looping structure



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]
```

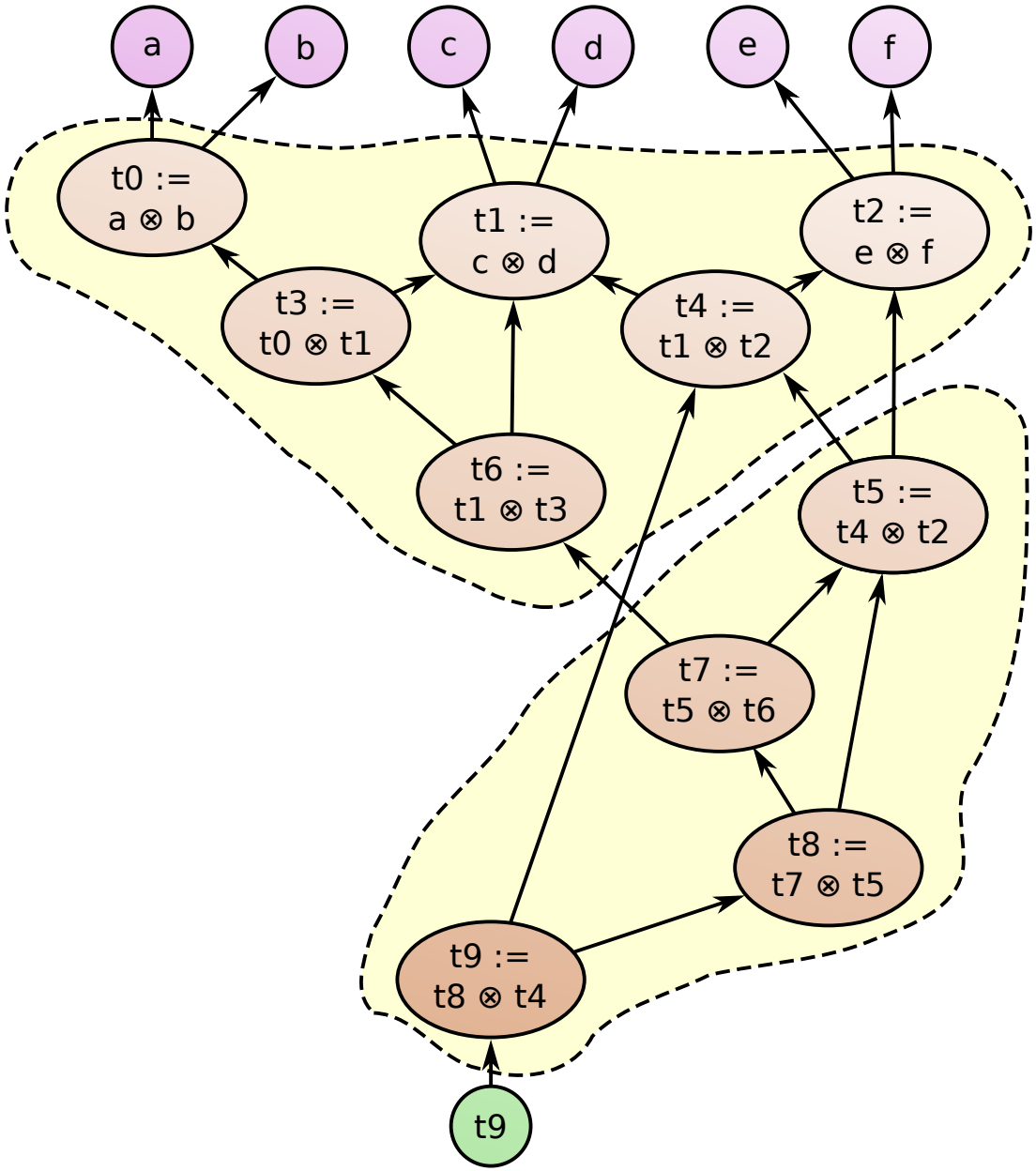
Freedom to choose the looping structure



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
```

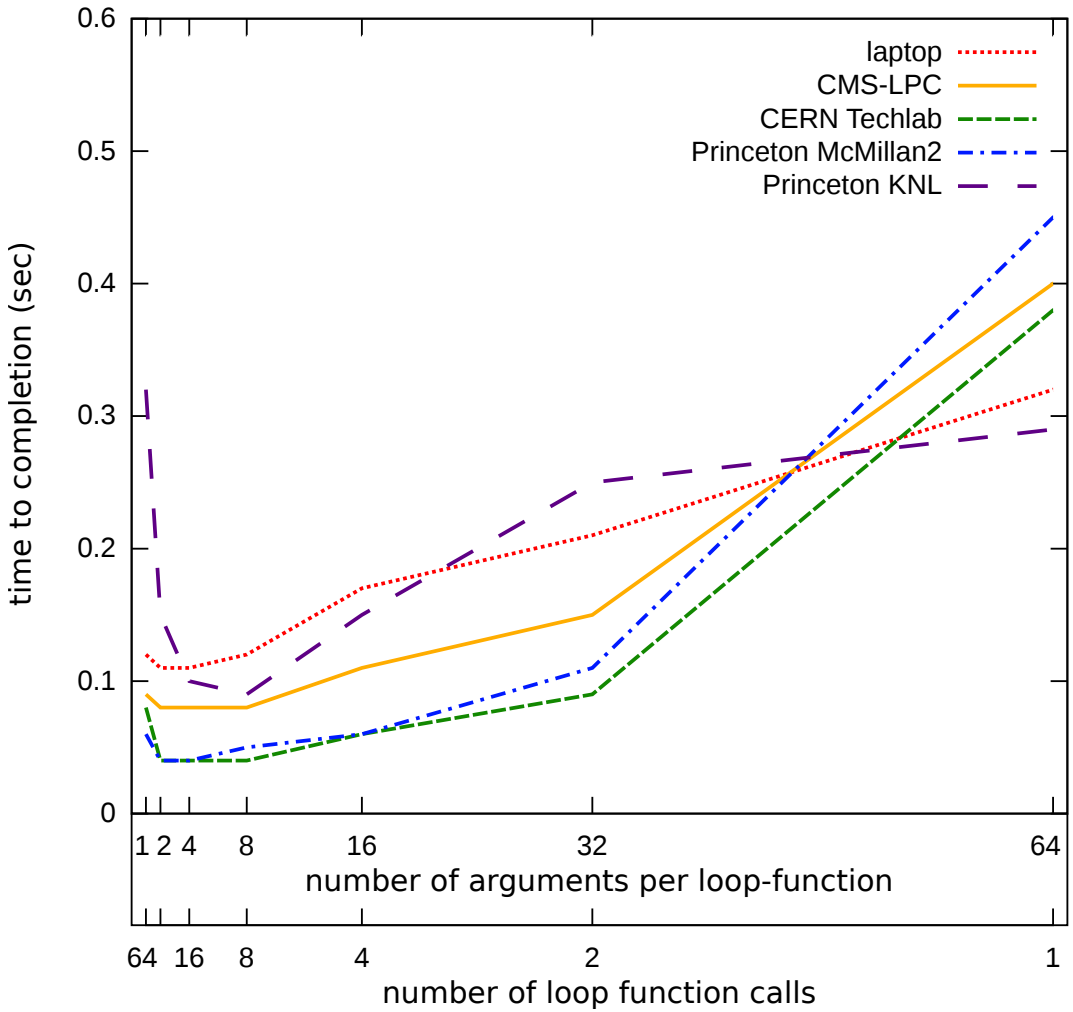
Note that this changes which quantities are arrays and which are scalars.

What are the trade-offs?

Assuming the bottleneck to be memory bandwidth (usually true), more loops:

- increases number of memory passes and
- sometimes decreases number of arrays to stride simultaneously.

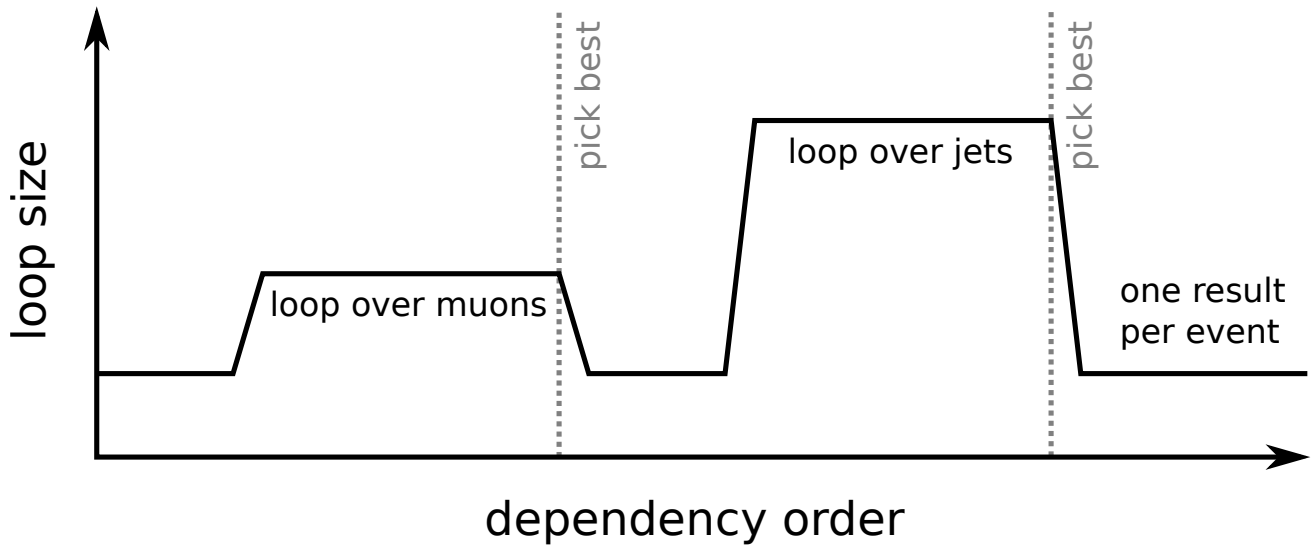
Test of splitting 1 loop over 64 variables into 64 loops over 1 variable reveals a sweet spot of about 2–32.



Sometimes you don't get to choose

Some vector operations have higher cardinality than others: e.g. a loop over jets has more steps than a loop over muons.

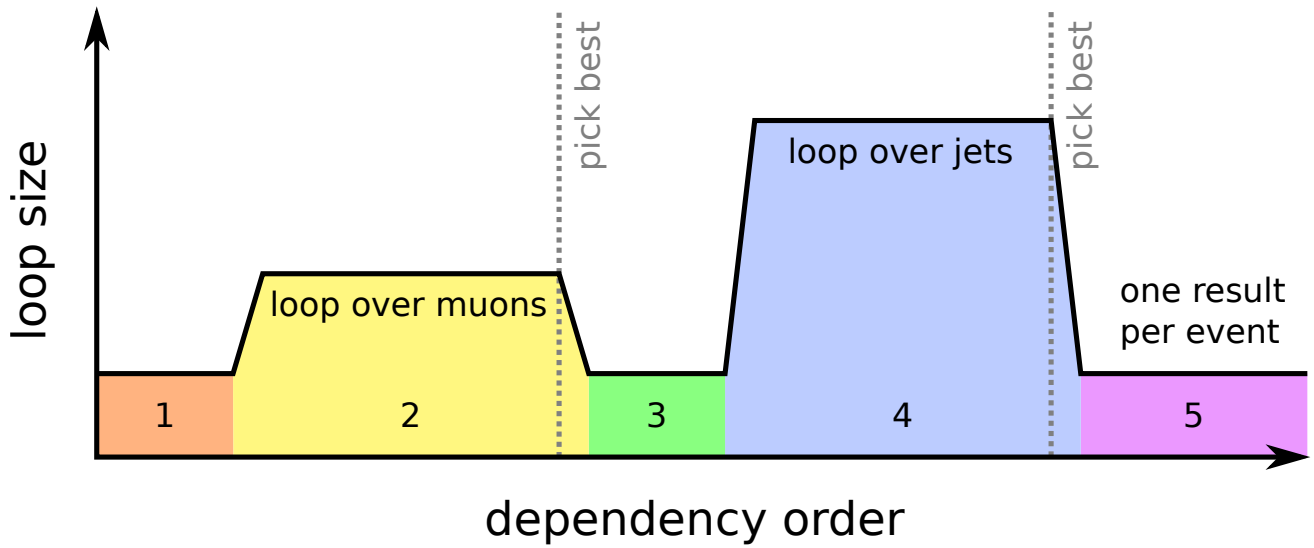
Operations of different cardinality can't be in the same loop, so Femtocode divides the dependency graph into "plateaus."



Sometimes you don't get to choose

Some vector operations have higher cardinality than others: e.g. a loop over jets has more steps than a loop over muons.

Operations of different cardinality can't be in the same loop, so Femtocode divides the dependency graph into "plateaus."

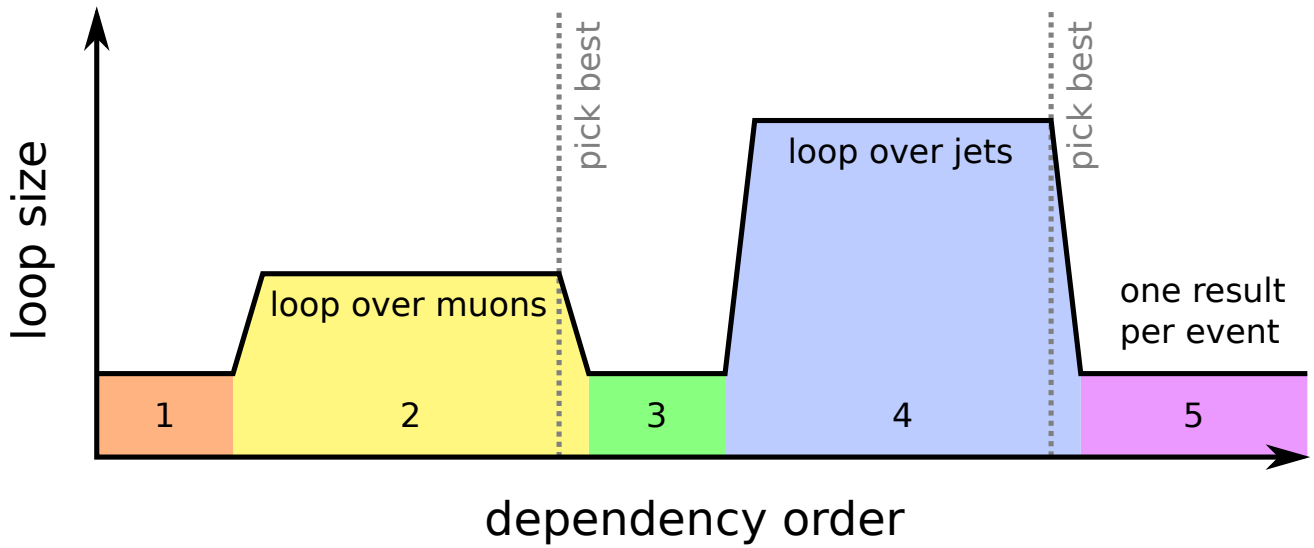


This cartoon example requires five loops (assuming each step strictly depends on the previous).

Sometimes you don't get to choose

Some vector operations have higher cardinality than others: e.g. a loop over jets has more steps than a loop over muons.

Operations of different cardinality can't be in the same loop, so Femtocode divides the dependency graph into "plateaus."

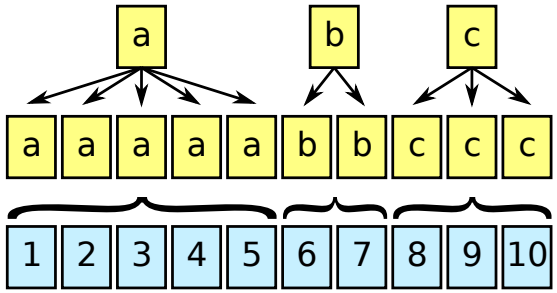


This cartoon example requires five loops (assuming each step strictly depends on the previous).

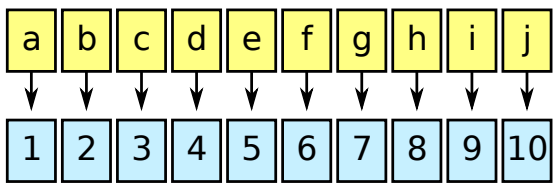
Our dimuon example naturally splits into two loops: one over muons (`muons[]@size`) and one over muons × muons (`#11@size`).

Three kinds of operations in each plateau

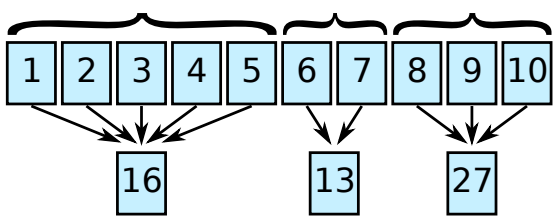
Explode: increase cardinality of one array so that it matches another. Determines the indexing of the loop, so must be first.



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.



Representing objects as arrays (1/2)

Muon object schema:

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

Physical representation:

```
arrays_in = {
    "muons[]-pt":    [31.0960, 9.7620, 8.1769, ...,
                      5.2730, 4.7240, 8.5879],     # (length 132274)
    "muons[]-phi":   [-0.4814, -0.1242, -0.1185, ...,
                      1.2469, -0.2067, -1.7541],   # (length 132274)
    "muons[]-eta":   [0.8816, 0.9243, 0.9226, ...,
                      -0.9911, 0.9532, -0.2635],   # (length 132274)
    "muons[]@size": [7, 1, 4, ..., 4, 0, 1]}       # (length 48131)
```

Dimuon run produces:

```
masses = collection(collection(union(null, real(0, almost(inf)))))
```

```
arrays_out = {
    "#34":          [0.2113, 6.2386, 5.7978, ...,
                     13.1108, 0.2113, 0.2113],    # (length 584642)
    "#11@size":     [7, 7, 7, ..., 0, 1, 1]}      # (length 180405)
```

Representing objects as arrays (2/2)

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.

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`

We know whether a number in the size array refers to the size of an outer collection or an inner collection from a stack of countdowns.

Code generated for recursive counters

a fully general example: `"xss.map(xs => xs.map(x => ys.map(y => x + y)))"`

```
entry = 0                    # entry index
deepi = 0                    # depth of collection
countdown = [0, 0, 0]        # stack of indexes
x_skip = [False, False]      # handling zero x_size
y_skip = [False]             # handling zero y_size

while entry < numEntries:    # master loop
    if deepi != 0:
        countdown[deepi - 1] -= 1

    if deepi == 0:           # xss.map(xs => ...)
        x_index[1] = x_index[0]
        countdown[deepi] = x_size[x_index[1]]
        x_index[1] += 1

        if countdown[deepi] == 0:
            x_skip[0] = True
            countdown[deepi] = 1
        else:
            x_skip[0] = False

    elif deepi == 1:         # xs.map(x => ...)
        x_index[2] = x_index[1]
        if not xskip[0]:
            countdown[deepi] = x_size[x_index[2]]
            x_index[2] += 1

        if countdown[deepi] == 0:
            x_skip[1] = True
            countdown[deepi] = 1
        else:
            x_skip[1] = False

    elif deepi == 2:         # ys.map(y => ...)
        y_index[1] = y_index[0]
        countdown[deepi] = y_size[y_index[1]]
        y_index[1] += 1

        if countdown[deepi] == 0:
            y_skip[0] = True
            countdown[deepi] = 1
        else:
            y_skip[0] = False

    elif deepi == 3:         # body of loop
        deepi -= 1

        if not x_skip[0] and not x_skip[1] \
           and not y_skip[0]:
            # put "x + y" into output array

    deepi += 1

    while deepi != 0 and countdown[deepi - 1] == 0:
        deepi -= 1           # "closing parentheses"

        if deepi == 0:
            x_index[0] = x_index[1]
            y_index[0] = y_index[1]

        elif deepi == 1:
            x_index[1] = x_index[2]

    if deepi == 0:     # master loop iterates through
        entry += 1     # deepest nesting level
```

Features of the event loop

- JIT-compiled for the specific nesting observed in query.
- Never allocates memory at runtime.
- Always two nested while-loops; the second only pops out of the stack (could be replaced by JIT-compiled if-statements).
- Walk through data is controlled by stacks of fixed depth (already replaced by JIT-compiled stack variables; 30% speedup).
- Memory access pattern is contiguous and usually forward, though it sometimes jumps backward to emulate loops like

```
muons.map(mu1 => muons.map(mu2 => ...))
```

- Open question: would a version of this using recursion, rather than a single loop with stacks, be faster?
- Generated as Python code (previous page), compiled by LLVM into native machine code. (Easier to test in Python.)

Why size arrays instead of runtime objects?

1. To help LLVM and the hardware optimize memory bandwidth.

Simple operation on 806177 jet p_T values (6.15 MB):	
3 ms	no-frills loop in C
7 ms	Numpy's implementation
14 ms	full generality Femtocode event loop
24 ms	allocating C++ objects on stack and iterating
64 ms	allocating C++ objects on heap, iterating, deleting
518 ms	`TTree::Draw` with `TTreeCache`
41900 ms	CMSSW EDAnalyzer (disk access)

(Note: Femtocode should ultimately resemble the no-frills loop in C. There's work to be done.)

Why size arrays instead of runtime objects?

diana hep

1. To help LLVM and the hardware optimize memory bandwidth.

   Simple operation on 806177 jet p_T values (6.15 MB):

3 ms	no-frills loop in C
7 ms	Numpy's implementation
14 ms	full generality Femtocode event loop
24 ms	allocating C++ objects on stack and iterating
64 ms	allocating C++ objects on heap, iterating, deleting
518 ms	`TTree::Draw` with `TTreeCache`
41900 ms	CMSSW EDAnalyzer (disk access)

   (Note: Femtocode should ultimately resemble the no-frills loop in C. There's work to be done.)

2. With no event boundaries in the data arrays, the "flat functions" perfectly satisfy the criteria for GPU acceleration.

   Thus, we could automatically translate high-level code on physics objects into well-optimized GPU kernels!

Using ROOT functions in Femtocode

```
########## 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
```

Distributed server

Scaling up to full datasets

Exploratory data analysis requires turn-around times on human timescales: seconds at most. If a query server takes much longer than this, physicists will go back to private skims.

Scaling estimates for one query:

- Typically use a dozen or so samples, totaling $\mathcal{O}(10\ \text{TB})$.
- Every query runs over all events, but a single query rarely uses 1% of the *columns.* (Popularity distribution is steep.)
- In this early implementation of Femtocode, the worst query response times were 30 ms/MB.
- Implies 3000 core-sec for that query: 3 seconds for 1000 cores.

Scaling up to full collaborations

Scaling estimates for multiple users:

- Most analyses have significantly overlapping needs. Evidence: home-grown skimming frameworks (Bacon, Pandas, Cms3, TreeMaker) select the same 10% of CMS MiniAOD.
- File I/O is more expensive than processing: ∼40 ms/MB versus ∼2 ms/MB. Everyone wins if users *share* cache.
- 10% of 10 TB of samples is 1 TB, which easily fits in RAM on a cluster of 1000 cores (hard to fit on one user's machine).
- Short-lived queries are less likely to use resources at the same time, so shortening latency also reduces contention.

The parameters of the final system depend on the hardware allocated for it, but improving software can steepen the performance per price.

Dividing the problem

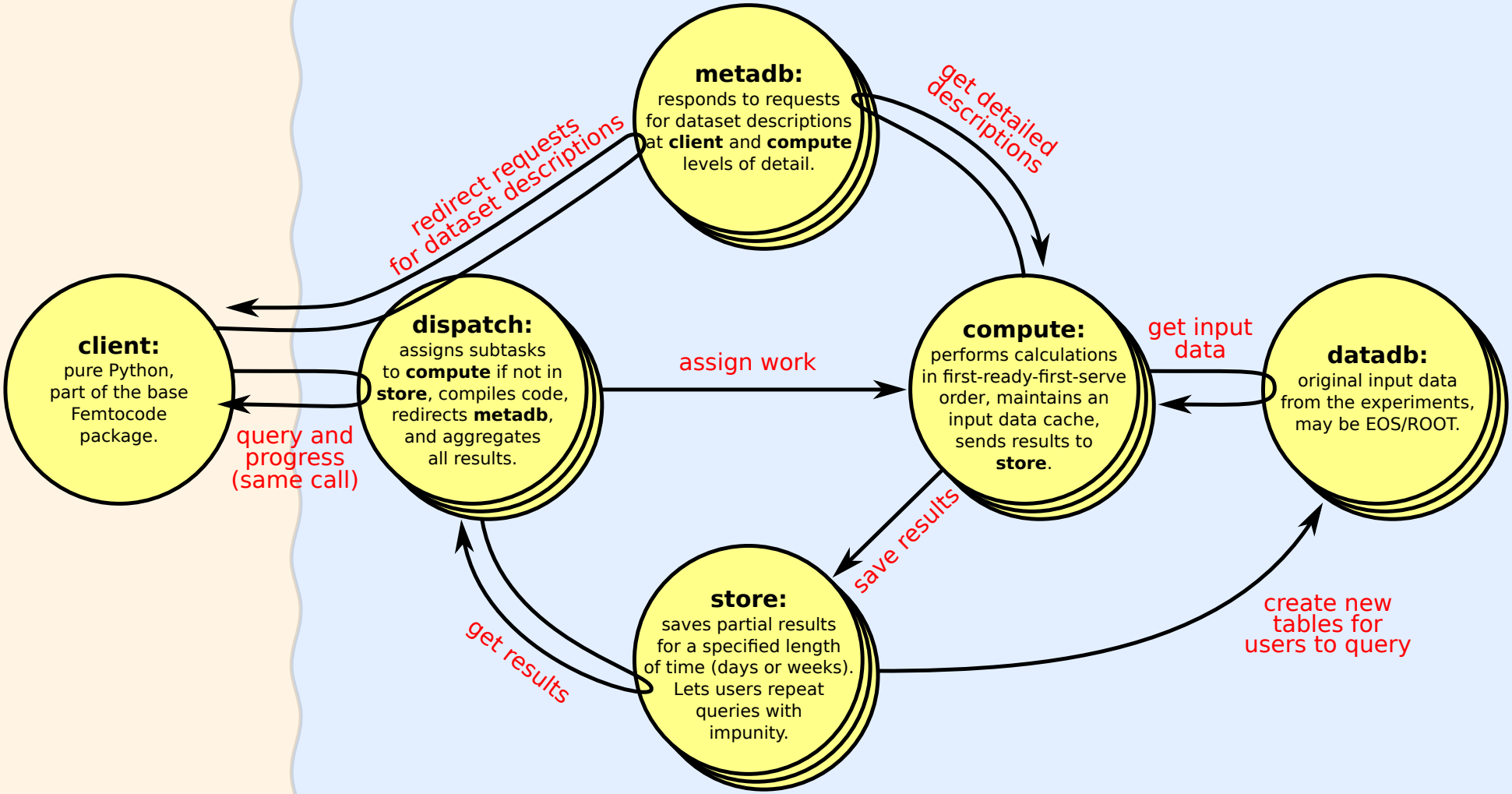
Femtocode's design philosophy is to do work up-front to streamline the event loop. In the distributed server, managing subtasks is part of this up-front work. Time to completion could be summarized as

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

- 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.

Distributed system layout: as implemented



Not a batch system

Although incoming jobs are scattered to compute nodes, computed in parallel, and then gathered, this differs from a batch system in important ways.

1. No "job id" or attempt to send results back to the user.

   Identical queries on the same dataset will always yield identical results, so jobs are identified by a hash of the query itself.

   Client polls for updates and may break/reestablish connection before it's done. Dispatch checks the "store" for partial results, rather than re-running.

   Therefore, when users refresh their analysis scripts or run tutorial examples, they don't stress the computation engine.

2. Subtasks are assigned to "compute" nodes based on what data they need (hash of input column names). That way, any cached input will be local to that node.

Loosely coupled, resistant to failure

Each circle on the diagram is a collection of identical nodes, none of which are single points of failure.

- If a "compute" node disappears, the hash-assignment function has a series of fallbacks.
- The "dispatch" nodes are stateless; they can be load-balanced.
- Only the "store" persistently holds results; it's a MongoDB instance with appropriate partitioning and replication.
- The datasets in "metadb" and "datadb" are treated as immutable artifacts. New datasets may be created (with version numbers), but not changed in-place.

Conclusions

Conclusions

Progress on all three aspects of the Femtocode query server project.

- Langauge/Compiler: starting to compute meaningful quantities. Parser and type-checker are mature.
- Execution engine: problem of how to compute general "explosions" is under control. Compiling with LLVM and even serializing compiled functions for remote execution. Factors of several from optimal performance.
- Distributed server: working prototype passes data through components, returns results as it should. Have not attempted to scale up.

Mature enough that there may be subprojects to split off. Ask me if you're interested!