Executing code on columnar data: the translation problem and formats that help

Jim Pivarski

Princeton – DIANA

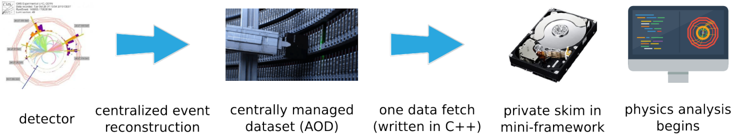
February 8, 2017

Why I'm interested in columnar data

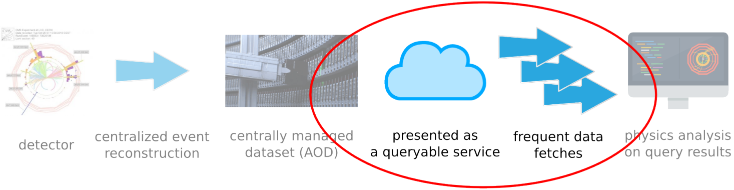
I'm working on a query language and database server to aggregate large samples of HEP data on the fly.

Purpose: to eliminate the need for private skims in most situations.

to replace



with



Why I'm interested in columnar data

The query language, Femtocode, plays a similar role as TTreeFormula:

- a high-level language for the physicist
- usually for filling a histogram (so query responses are small)
- but generally useful for transforming one dataset into another.

However, it's a full-fledged language with assignments and user-defined functions, so that it can encompass a larger part of the analysis.

(I've examined SQL, LINQ, and others, and they are not sufficient. I would use a standard if I could.)

Why I'm interested in columnar data

The essential feature of Femtocode is that it can transform complex structure-manipulations, which would ordinarily have to be performed in object-oriented code, into a series of vectorized kernels.

It operates on columns.

Data reduction without objects

Example:

```
hist = dataset..bin(100, 0, 50, """
    muons.map(m => sqrt(m.px**2 + m.py**2)).max()
""")
```

transforms to

1. Pass over all muons, computing $\sqrt{{p_x}^2 + {p_y}^2}$ on all muons, ignoring event boundaries.
2. Find the maximum such value for each event.
3. Bin those events and fill the histogram.

rather than

1. Loop over events:
   - 1.1 Loop over muons:
     - 1.1.1 Compute $\sqrt{{p_x}^2 + {p_y}^2}$ for each.
   - 1.2 Fill a histogram with the maximum.

Scope of computability

Three types of data transformations:

Flat: apply N-argument function to each element of N aligned arrays.

Known in the Numpy community as a "ufunc."

Explode: emulate (nested) for loops by replicating data in one array so that it becomes aligned with another array.

Reduce: emulate reducer functions (sum, mean, max...) by combining elements of an array so that it becomes aligned with an outer level of structure.

Scope of computability

Three types of data transformations:

Flat: apply N-argument function to each element of N aligned arrays.

Known in the Numpy community as a "ufunc."

Explode: emulate (nested) for loops by replicating data in one array so that it becomes aligned with another array.

Reduce: emulate reducer functions (sum, mean, max...) by combining elements of an array so that it becomes aligned with an outer level of structure.

What's missing?

"Repeat until convergence." Whatever determines "convergence" may be different for each element of the array: they'd all wait for the last one, anyway.

Flat transformations

The majority of steps in a typical calculation are flat:

```
double in[ZILLION];
double out[ZILLION];

for (int i = 0;  i < ZILLION;  i++)
  out[i] = flat_operation(in[i]);
```

- Compilation with `-O3` does auto-vectorization if possible (depends on `flat_operation`).
- Easiest form for CPU to prefetch memory and/or pipeline operations.
- This form is also ideal for GPU calculations.
- There is a standard for functions like this: Numpy's ufunc is widely used among scientific libraries.
  - Easy way for a user to add functions to the language!

ROOT functions can be ufuncs, too

```python
import ctypes, numpy, numba

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

# compile to pure-C ufunc
@numba.vectorize(["f8(f8, f8)"], nopython=True)
def chi2_ufunc(p, ndf):
    return chi2_ctypes(p, ndf)

p = numpy.random.uniform(0, 1, int(1e6)) # million random numbers
result = chi2_ufunc(p, 100)              # call ufunc on all of them
# 3.22 seconds

import ROOT
result = [ROOT.TMath.ChisquareQuantile(pi, 100) for pi in p]
# 9.32 seconds
```

(Performance comparison is just to show that the ufunc computes `ChisquareQuantile` in C, not in Python. Simpler functions show a more dramatic difference.)