Spark-like and query-like analysis systems and tools

(what works, what doesn't work, what's needed, and what we're building to fill that need)

Jim Pivarski

Princeton University – DIANA-HEP

March 27, 2018

“Analysis system” is a data access optimization scenario

The following are often true about end-user data analysis:

- Only need a small fraction of the event and particle attributes, like a few dozen.
  (A handful of trigger flags, details on two or three particle types, maybe a veto on another’s kinematics, but nowhere close to the thousands of available attributes.)
- Need to look at all events, if only to reject them on the basis of a few attributes.
- Frequent, repeated process; code to run not known in advance; exploratory.

“Analysis system” is a data access optimization scenario

The following are often true about end-user data analysis:

- Only need a small fraction of the event and particle attributes, like a few dozen.
  (A handful of trigger flags, details on two or three particle types, maybe a veto on another’s kinematics, but nowhere close to the thousands of available attributes.)
- Need to look at all events, if only to reject them on the basis of a few attributes.
- Frequent, repeated process; code to run not known in advance; exploratory.

Today’s pipelines were designed for a different optimization scenario: reconstruction.

“Analysis system” is a data access optimization scenario

The following are often true about end-user data analysis:

- Only need a small fraction of the event and particle attributes, like a few dozen.
  (A handful of trigger flags, details on two or three particle types, maybe a veto on another’s kinematics, but nowhere close to the thousands of available attributes.)
- Need to look at all events, if only to reject them on the basis of a few attributes.
- Frequent, repeated process; code to run not known in advance; exploratory.

Today’s pipelines were designed for a different optimization scenario: reconstruction.

Consider this request: “Please submit a GRID job to plot the muon p_T spectrum. Then we’ll think about what we want to look at next.” Is that unreasonable?

An example of what I have in mind: Google BigQuery

Google BigQuery

COMPOSE QUERY

Query History

Job History

Filter by ID or label

HEPQuery

No datasets found in this project.

Please create a dataset or select a new project from the menu above.

Public Datasets

- bigquery-public-data:hacker_news
- bigquery-public-data:noaa_gsod
- bigquery-public-data:samples
- bigquery-public-data:usa_names
- gdelt-bq:hathitrustbooks
- gdelt-bq:internetarchivebooks
- lookerdata:cdc
- nyc-tlc:green
- nyc-tlc:yellow

New Query

Query Editor | UDF Editor

```sql
SELECT timestamp, country_code, file.filename, file.version, file.type, url, details.distro.name, details.distro.version, details.system.name, details.system.release, details.cpu
FROM TABLE_DATE_RANGE(
  [the-psf:pypi.downloads],
  TIMESTAMP("2017-01-20"),
  CURRENT_TIMESTAMP()
)
WHERE file.project="uproot" and details.installer.name == "pip" and details.distro.name == "Raspbian GNU/Linux"
```

Ctrl + Enter: run query, Tab or Ctrl + Space: autocomplete.

RUN QUERY | Save Query | Save View | Format Query | Show Options

Query complete (30.2s elapsed, 1.82 TB processed)

Results | Details

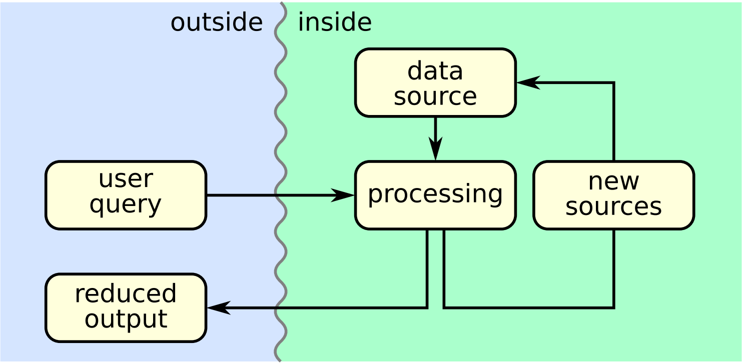
Download as CSV | Download as JSON | Save as Table | Save to Google Sheets

Row	timestamp	country_code	file_filename	file_version	file_type	url	details_distro_name	details_distro_version	det
1	2017-11-24 23:26:53.000 UTC	null	uproot-2.0.2.tar.gz	2.0.2	sdist	/packages/f4/06/91b0cc7d9b5ef9cb071e72b8ce77195baa22a6c99c431680a144ab858026/uproot-2.0.2.tar.gz	Raspbian GNU/Linux	9	Linu
2	2017-11-24 23:02:03.000 UTC	null	uproot-2.0.2.tar.gz	2.0.2	sdist	/packages/f4/06/91b0cc7d9b5ef9cb071e72b8ce77195baa22a6c99c431680a144ab858026/uproot-2.0.2.tar.gz	Raspbian GNU/Linux	9	Linu
3	2017-12-11 21:51:53.000 UTC	null	uproot-2.5.9.tar.gz	2.5.9	sdist	/packages/7d/b0/ecf6134c205e2678ca28fa7493303576313221638 49c017b2fd635822291/uproot-2.5.9.tar.gz	Raspbian GNU/Linux	8	Linu
4	2017-11-30 14:15:25.000 UTC	null	uproot-2.1.5.tar.gz	2.1.5	sdist	/packages/96/1c/f246d9beae81fbad3fc1d0bd675a0516b4286c23b9afefe92f3099bf5f6d/uproot-2.1.5.tar.gz	Raspbian GNU/Linux	8	Linu
5	2017-11-30 13:35:34.000 UTC	null	uproot-2.3.3.tar.gz	2.3.3	sdist	/packages/5d/8b/dc6486334876c81c23034e60dd836fe9b2bf7eb4e5b39df1a83e1e8aaa23/uproot-2.3.3.tar.gz	Raspbian GNU/Linux	8	Linu
6	2018-01-03 23:50:16.000 UTC	null	uproot-2.5.15.tar.gz	2.5.15	sdist	/packages/7a/1f/c4fcb3b32809f45c03c3d7adfb1051176b6ad35c104040fc9fa38f47b44c/uproot-2.5.15.tar.gz	Raspbian GNU/Linux	8	Linu
7	2018-01-03 16:47:34.000 UTC	null	uproot-2.5.14.tar.gz	2.5.14	sdist	/packages/e8/65/858fd40c433c9a1034fa3062897ff3aa19a58c2d80d596fdbbf6df573c2f/uproot-2.5.14.tar.gz	Raspbian GNU/Linux	8	Linu
8	2017-11-04 02:11:29.000 UTC	null	uproot-1.0.1.tar.gz	1.0.1	sdist	/packages/d3/50/4af1773d449a454ee31e20278ec114bceba1bc473b636a11685d40b6552f/uproot-1.0.1.tar.gz	Raspbian GNU/Linux	8	Linu
9	2017-11-04 02:11:54.000 UTC	null	uproot-1.4.2.tar.gz	1.4.2	sdist	/packages/e1/9b/6ae6cc054af7e61fecc7cfb45b7a18c0f3c06dcfd1a596ca2cb844f84a93/uproot-1.4.2.tar.gz	Raspbian GNU/Linux	8	Linu
10	2017-11-04 02:12:01.000 UTC	null	uproot-1.6.0.tar.gz	1.6.0	sdist	/packages/98/0c/7a7f934b1e7a9a302e8ef16edc030ba17e5759141ed9e86aaefbb0464734/uproot-1.6.0.tar.gz	Raspbian GNU/Linux	8	Linu
11	2017-11-04 02:11:55.000 UTC	null	uproot-1.5.3.tar.gz	1.5.3	sdist	/packages/60/14/bfa043c970b66b6f73fe0765ffdeec5964338c2a29301cc2a1c7740be209/uproot-1.5.3.tar.gz	Raspbian GNU/Linux	8	Linu
12	2017-11-04 02:11:55.000 UTC	null	uproot-1.5.0.tar.gz	1.5.0	sdist	/packages/b7/12/3fd299b2fd9c2076d491933ac7a1177ade4a0d715fbc56beb8e14c91cf66/uproot-1.5.0.tar.gz	Raspbian GNU/Linux	8	Linu
13	2017-11-04 02:11:27.000 UTC	null	uproot-1.0.0.tar.gz	1.0.0	sdist	/packages/5c/0a/7ca1f587d460cb8fe70306a94308822b2cce5bf176cd4f0879246950c73f/uproot-1.0.0.tar.gz	Raspbian GNU/Linux	8	Linu
14	2017-11-04 02:11:29.000 UTC	null	uproot-1.2.0.tar.gz	1.2.0	sdist	/packages/c0/e1/c249b359c0dfce204968f6fefff84813199b77b66a43b36727c4d2ed60f4/uproot-1.2.0.tar.gz	Raspbian GNU/Linux	8	Linu
15	2017-11-04 02:11:39.000 UTC	null	uproot-1.3.0.tar.gz	1.3.0	sdist	/packages/7e/6d/f070aff4821b78a8bf7d133922ad93540c82a01924473e1d77149d6fb9be/uproot-1.3.0.tar.gz	Raspbian GNU/Linux	8	Linu
16	2017-11-04 02:12:05.000 UTC	null	uproot-1.6.1.tar.gz	1.6.1	sdist	/packages/d7/39/8eadf6a705cd6b6b778fc2b067d920b0776bb1b3b11e3359f55d786640cb/uproot-1.6.1.tar.gz	Raspbian GNU/Linux	8	Linu
17	2017-11-04 02:11:40.000 UTC	null	uproot-1.3.1.tar.gz	1.3.1	sdist	/packages/1b/b2/c59d1916ca1aa631f2264114cbc4ac16a226ac705821a2ab48e6bb16a352/uproot-1.3.1.tar.gz	Raspbian GNU/Linux	8	Linu

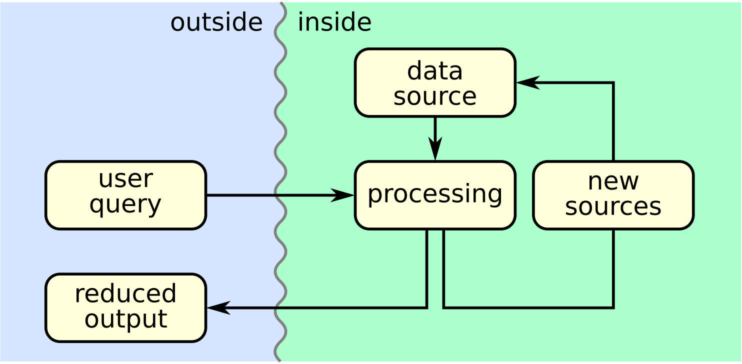
Table | JSON

First < Prev Rows 1 - 17 of 37 Next > Last

Basic block diagram

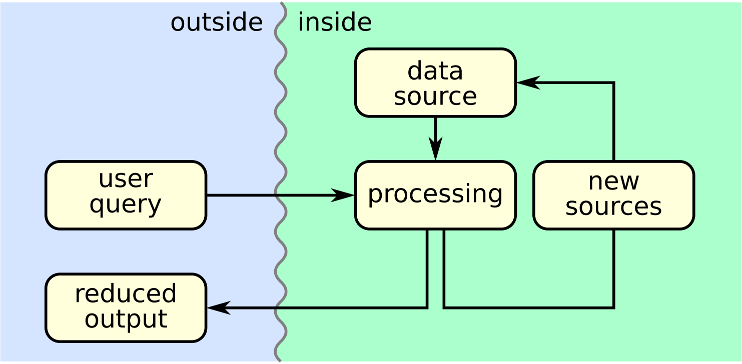


Basic block diagram



The "reduced output" must be small enough to download quickly (e.g. histograms or highly skimmed tables). If not, the system ceases to be "exploratory."

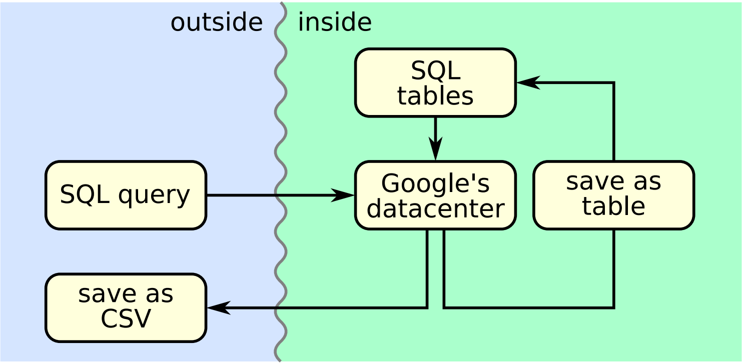
Basic block diagram



The "reduced output" must be small enough to download quickly (e.g. histograms or highly skimmed tables). If not, the system ceases to be "exploratory."

The "new sources" can be arbitrarily large and efficiently share overlapping data with the original sources because they live in the same system.

Basic block diagram



The “reduced output” must be small enough to download quickly (e.g. histograms or highly skimmed tables). If not, the system ceases to be “exploratory.”

The “new sources” can be arbitrarily large and efficiently share overlapping data with the original sources because they live in the same system.

So we can just use BigQuery, then?

No!!!

So we can just use BigQuery, then?

No!!!

- Reason #1: SQL. Simple HEP problems translate into complex SQL and moderate-to-complex HEP problems would be unreasonably difficult to express.

  Non-SQL languages in this problem space, such as SparkSQL's column expressions or Apache Drill's internal query language, are just as restrictive in the ways that matter.

No!!!

- Reason #1: SQL. Simple HEP problems translate into complex SQL and moderate-to-complex HEP problems would be unreasonably difficult to express.

  Non-SQL languages in this problem space, such as SparkSQL's column expressions or Apache Drill's internal query language, are just as restrictive in the ways that matter.

- Reason #2: Data model. The BigQuery paper ("Dremel"), Parquet & Drill (open source versions of the same), Apache Arrow, and SparkSQL all describe rich, nested data models sufficient to describe HEP events.

  However, *for every one of these,* you quickly encounter "not implemented yet" messages when you try to use them for HEP events.

So we can just use BigQuery, then?

No!!!

- Reason #1: SQL. Simple HEP problems translate into complex SQL and moderate-to-complex HEP problems would be unreasonably difficult to express.

  Non-SQL languages in this problem space, such as SparkSQL's column expressions or Apache Drill's internal query language, are just as restrictive in the ways that matter.

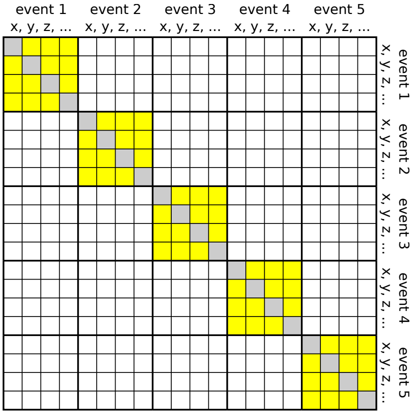
- Reason #2: Data model. The BigQuery paper ("Dremel"), Parquet & Drill (open source versions of the same), Apache Arrow, and SparkSQL all describe rich, nested data models sufficient to describe HEP events.

  However, *for every one of these,* you quickly encounter "not implemented yet" messages when you try to use them for HEP events.

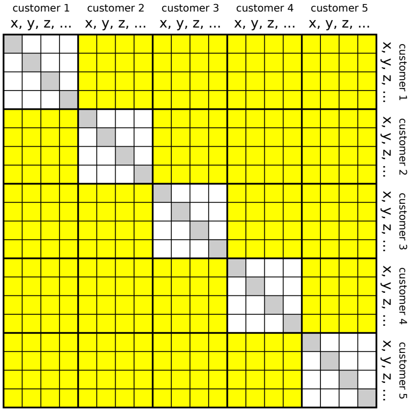
- Reason #3: User interface. The web form is fun, but we need queries embedded in an interactive, programmable environment with plotting and statistics libraries: ROOT or Python or both (probably both).

HEP analysis is simpler

HEP functions never have to cross events.



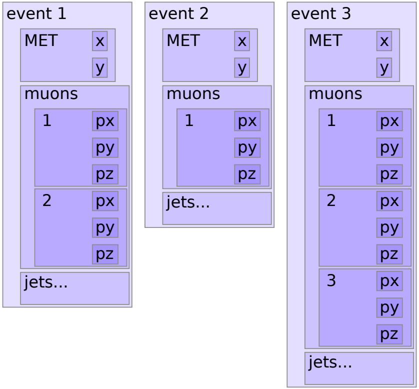
Common elsewhere: e.g. market basket analysis.



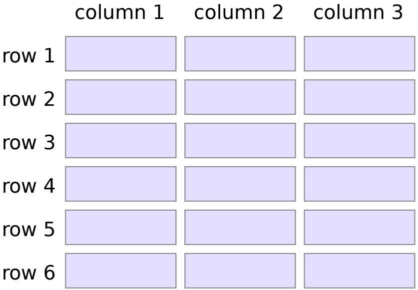
HEP is both simpler and more complex than relational analytics

HEP analysis is more complex

HEP data are variable-length, nested data structures, and we typically need to loop over combinations of particles.



In many fields, data are not considered ready for analysis until they're in a tabular form. (Earlier steps are called "tidying.")



HEP is both simpler and more complex than relational analytics

	independent events	all-to-all shuffle
tabular data	basic spreadsheets	relational analytics
variable-length, nested data	HEP analysis	graph analytics

What should our data source be?

Identically typed, variable-length, nested data can be split into columns:

- All attribute values at a given level of hierarchy are stored together and may be retrieved independently of the rest.
- Good for reading only the dozen interesting attributes.
- ROOT has been doing this for years.

What should our data source be?

Identically typed, variable-length, nested data can be split into columns:

- All attribute values at a given level of hierarchy are stored together and may be retrieved independently of the rest.
- Good for reading only the dozen interesting attributes.
- ROOT has been doing this for years.

But we could take this further, the way fast databases have:

- **Minimize runtime:** data analysis functions have low arithmetic intensity; don't spend time creating objects that will be used only once!

What should our data source be?

Identically typed, variable-length, nested data can be split into columns:

- All attribute values at a given level of hierarchy are stored together and may be retrieved independently of the rest.
- Good for reading only the dozen interesting attributes.
- ROOT has been doing this for years.

But we could take this further, the way fast databases have:

- **Minimize runtime:** data analysis functions have low arithmetic intensity; don't spend time creating objects that will be used only once!
- **Minimize storage:** new versions of a dataset can share (as in "symlink") most attributes with the old version.

What should our data source be?

Identically typed, variable-length, nested data can be split into columns:

- All attribute values at a given level of hierarchy are stored together and may be retrieved independently of the rest.
- Good for reading only the dozen interesting attributes.
- ROOT has been doing this for years.

But we could take this further, the way fast databases have:

- **Minimize runtime:** data analysis functions have low arithmetic intensity; don't spend time creating objects that will be used only once!
- **Minimize storage:** new versions of a dataset can share (as in "symlink") most attributes with the old version.
- **Share cache:** *column* popularity distribution is steeper than the *file* popularity distribution; most popular columns could remain in memory for all users.

What should our data source be?

Identically typed, variable-length, nested data can be split into columns:

- All attribute values at a given level of hierarchy are stored together and may be retrieved independently of the rest.
- Good for reading only the dozen interesting attributes.
- ROOT has been doing this for years.

But we could take this further, the way fast databases have:

- **Minimize runtime:** data analysis functions have low arithmetic intensity; don't spend time creating objects that will be used only once!
- **Minimize storage:** new versions of a dataset can share (as in "symlink") most attributes with the old version.
- **Share cache:** *column* popularity distribution is steeper than the *file* popularity distribution; most popular columns could remain in memory for all users.
- **Avoid touching disk:** sorted partitions do not need to be fully read if a cut is being applied to the sorted variable (most likely p_T).

OAMap (https://github.com/diana-hep/oamap)

Object Array Map: columnar, hierarchical data as runtime objects

OAMap (https://github.com/diana-hep/oamap)

- √ Translates object-oriented (Python) code into nothing but array operations.

OAMap (https://github.com/diana-hep/oamap)

- √ Translates object-oriented (Python) code into nothing but array operations.
- √ Extends Numba (Python compiler) to generate native machine bytecode (fast).

Object Array Map: columnar, hierarchical data as runtime objects

OAMap (https://github.com/diana-hep/oamap)

- √ Translates object-oriented (Python) code into nothing but array operations.
- √ Extends Numba (Python compiler) to generate native machine bytecode (fast).
- √ Arrays can come from anywhere: ROOT (through uproot), raw data files, remote network calls, HDF5, etc.

  The same object can have attributes served from all of the above, so an official dataset can be served from ROOT files while a user's modification (cuts or partial calculations applied) can be these ROOT files augmented by raw arrays.

Object Array Map: columnar, hierarchical data as runtime objects

OAMap (https://github.com/diana-hep/oamap)

- $\surd$ Translates object-oriented (Python) code into nothing but array operations.
- $\surd$ Extends Numba (Python compiler) to generate native machine bytecode (fast).
- $\surd$ Arrays can come from anywhere: ROOT (through uproot), raw data files, remote network calls, HDF5, etc.

  The same object can have attributes served from all of the above, so an official dataset can be served from ROOT files while a user's modification (cuts or partial calculations applied) can be these ROOT files augmented by raw arrays.
- $\surd$ Array fetching is through an arbitrary dict-like object: may defer to disk, network, or per-column cache.

Object Array Map: columnar, hierarchical data as runtime objects

OAMap (https://github.com/diana-hep/oamap)

- √ Translates object-oriented (Python) code into nothing but array operations.
- √ Extends Numba (Python compiler) to generate native machine bytecode (fast).
- √ Arrays can come from anywhere: ROOT (through uproot), raw data files, remote network calls, HDF5, etc.

  The same object can have attributes served from all of the above, so an official dataset can be served from ROOT files while a user's modification (cuts or partial calculations applied) can be these ROOT files augmented by raw arrays.
- √ Array fetching is through an arbitrary dict-like object: may defer to disk, network, or per-column cache.
- √ OAMap has enough indirection that different particle types can be *independently sorted* by their own p_Ts. Thus, a filter like "muon $p_T > X$ and jet $p_T > Y$" may touch disk for only half the muon data and half the jet data.

OAMap example

```
>>> import uproot
>>> import oamap.source.root

>>> url = "http://scikit-hep.org/uproot/examples/HZZ.root"
>>> events = uproot.open(url)["events"].oamap()
>>> events.schema.content["muons"].show()
List(
  starts = 'NMuon',      # schema maps object attributes to array names
  stops = 'NMuon',
  content = Record(      # at all levels of nesting
    fields = {
      'px': Primitive(dtype('float32'), data='Muon_Px'),
      'py': Primitive(dtype('float32'), data='Muon_Py'),
      'pz': Primitive(dtype('float32'), data='Muon_Pz'),
      'energy': Primitive(dtype('float32'), data='Muon_E'),
      'charge': Primitive(dtype('int32'), data='Muon_Charge'),
      'isolation': Primitive(dtype('float32'), data='Muon_Iso')
    }))
```

OAMap example

The dataset looks like a nested Python list.

```
>>> events
[<Event at index 0>, <Event at index 1>, <Event at index 2>, ...,
 <Event at index 2418>, <Event at index 2419>, <Event at index 2420>]

>>> events[0].muons
[<Muon at index 0>, <Muon at index 1>]

>>> [x.px for x in events[0].muons]
[-52.899456, 37.73778]
```

But it is generated on demand from arrays.

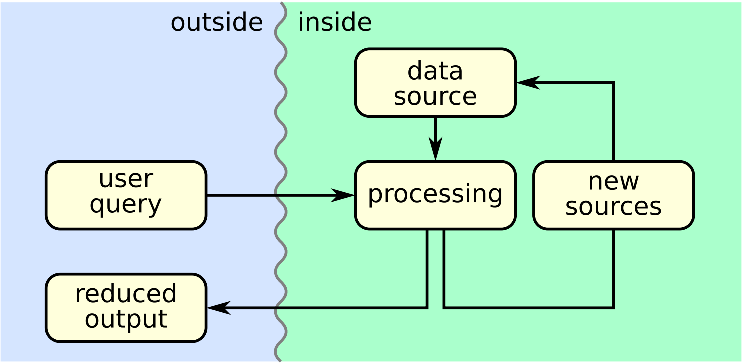
```
"NMuon":   array([2, 1, 2, ..., 1, 1, 1], dtype=int32)
"Muon_Px": array([-52.899456,  37.73778,  -0.81645936, ...,
                  -29.756786, 1.1418698,  23.913206  ], dtype=float32)
```

OAMap example

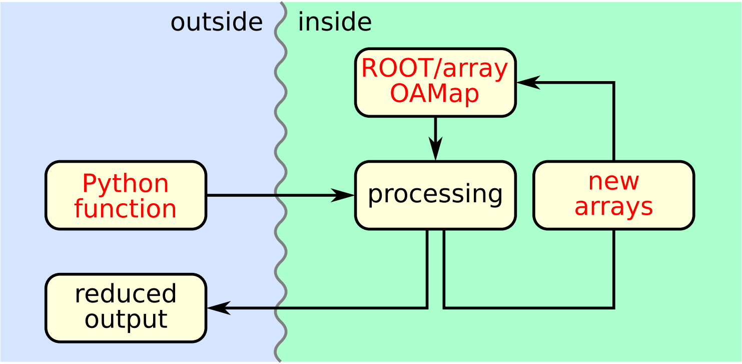
It can also be included in compiled code with no change in syntax.

```
>>> import numpy
>>> import numba
>>> import oamap.compiler
>>> @numba.njit                        # declares the following function to be compiled
... def compute(events, out):
...     i = 0
...     for event in events:           # "event" and "event.muons" are a compiler fiction
...         if len(event.muons) == 2:
...             mu1, mu2 = event.muons[0], event.muons[1]
...             px = mu1.px + mu2.px
...             py = mu1.py + mu2.py
...             pz = mu1.pz + mu2.pz
...             energy = mu1.energy + mu2.energy
...             out[i] = sqrt(energy**2 - px**2 - py**2 - pz**2)
...             i += 1

>>> out = numpy.empty(1371)
>>> compute(events, out)               # compilation and array-fetching happen on first call
>>> out
array([90.22780609, 74.74654388, 89.75765991, ..., 92.06494904,
       85.44384003, 75.96061707])
```

Current thinking. . .



Current thinking. . .



What should our reduced output be?

Simplest case: histograms, but that would get restrictive as analyses develop.

What should our reduced output be?

Simplest case: histograms, but that would get restrictive as analyses develop.

Here's an idea: query server returns Pandas DataFrames.

What should our reduced output be?

Simplest case: histograms, but that would get restrictive as analyses develop.

Here's an idea: query server returns Pandas DataFrames.

- Until recently, I've had the wrong idea about what a Pandas DataFrame is:
  I thought it was a TTree (set of events) with egregious limitations:
  - Tabular, with little support for variable-length, nested data.
  - Strictly in-memory. (Dask DataFrames implement the interface without this restriction.)

What should our reduced output be?

Simplest case: histograms, but that would get restrictive as analyses develop.

Here's an idea: query server returns Pandas DataFrames.

- Until recently, I've had the wrong idea about what a Pandas DataFrame is:
  I thought it was a TTree (set of events) with egregious limitations:
  - Tabular, with little support for variable-length, nested data.
  - Strictly in-memory. (Dask DataFrames implement the interface without this restriction.)
- I had been missing an important fact: most of Pandas's functionality is in its handling of *indexes*.
  - TTrees/events are indexed only by entry or run/event number.
  - Pandas indexes may be non-contiguous, non-numeric, intervals/durations, multi-component, . . . , and every operation maintains consistent indexes.

What should our reduced output be?

Simplest case: histograms, but that would get restrictive as analyses develop.

Here's an idea: query server returns Pandas DataFrames.

- Until recently, I've had the wrong idea about what a Pandas DataFrame is:
  I thought it was a TTree (set of events) with egregious limitations:
  - Tabular, with little support for variable-length, nested data.
  - Strictly in-memory. (Dask DataFrames implement the interface without this restriction.)
- I had been missing an important fact: most of Pandas's functionality is in its handling of *indexes*.
  - TTrees/events are indexed only by entry or run/event number.
  - Pandas indexes may be non-contiguous, non-numeric, intervals/durations, multi-component, . . . , and every operation maintains consistent indexes.
- Pandas has more in common with *histograms* than it does with *event sources*.

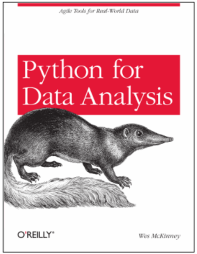
Just an example. . .

I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),
   ....:                  index=pd.Index(['Ohio', 'Colorado'], name='state'),
   ....:                  columns=pd.Index(['one', 'two', 'three'], name='number'))

In [95]: data
Out[95]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Using the `stack` method on this data pivots the columns into the rows, producing a Series:

```
In [96]: result = data.stack()

In [97]: result
Out[97]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
```



Pandas generalizes what we do with histograms

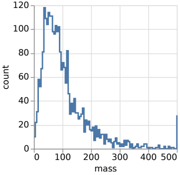
```python
import pandhist
# define bins in many dimensions; we'll think about how to plot later
muonhist = (pandhist
              .bin(100, 0, 500, "mass")
              .cut("q1*q2 < 0")
              .irrbin([0.2, 0.5], "mt1")
              .irrbin([0.2, 0.5], "mt2")
              .fillable())       # creates a fillable Pandas DataFrame

for muons, charge, mt2activity in uproot.iterate(
  "RA2Analysis/*.root", "TreeMaker2/PreSelection",
  ["Muons", "Muons_charge", "Muons_MT2Activity"], outputtype=tuple):
    for i in range(len(muons)):
        if len(charge[i]) == 2:
            mu1, mu2 = muons[i]
            q1,  q2  = charge[i]
            mt1, mt2 = mt2activity[i]
            # fill method has an argument for each variable
            muonhist.fill((mu1 + mu2).mass, q1, q2, mt1, mt2)
```

Pandas generalizes what we do with histograms

```
>>> muonhist
                                                        count   # in this example, count is
mass            q1*q2 < 0 mt1           mt2                     # the only column; the rest
[-inf, 0.0)     fail      [-inf, 0.2)   [-inf, 0.2)       0.0   # is a hierarchical index
                                        [0.2, 0.5)        0.0   # (mass, q1*q2 < 0, mt1, mt2)
                                        [0.5, inf)        0.0
                          [0.2, 0.5)    [-inf, 0.2)       0.0
                                        [0.2, 0.5)        0.0   # if we asked for weights,
                                        [0.5, inf)        0.0   # sumw and sumw2 would be
                          [0.5, inf)    [-inf, 0.2)       0.0   # separate columns
                                        [0.2, 0.5)        0.0
                                        [0.5, inf)        0.0
                pass      [-inf, 0.2)   [-inf, 0.2)       0.0   # if we asked for a profile,
                                        [0.2, 0.5)        0.0   # we'd get sum(y) and sum2(y)
                                        [0.5, inf)        0.0
                          [0.2, 0.5)    [-inf, 0.2)       0.0
                                        [0.2, 0.5)        0.0   # but most of the work is done
                                        [0.5, inf)        0.0   # by the hierarchical index
                          [0.5, inf)    [-inf, 0.2)       0.0
                                        [0.2, 0.5)        0.0
                                        [0.5, inf)        0.0
...                                                       ...
[1836 rows x 1 columns]
```

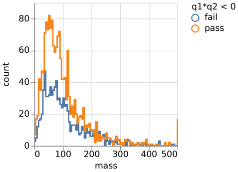
Pandas generalizes what we do with histograms

```
>>> pandhist.steps("mass").data(muonhist)
```



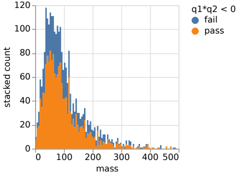
Pandas generalizes what we do with histograms

```
>>> pandhist.steps("mass").overlay("q1*q2 < 0").data(muonhist)
```



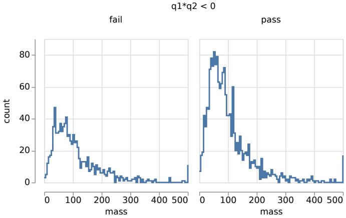
Pandas generalizes what we do with histograms

```
>>> pandhist.area("mass").stack("q1*q2 < 0").data(muonhist)
```



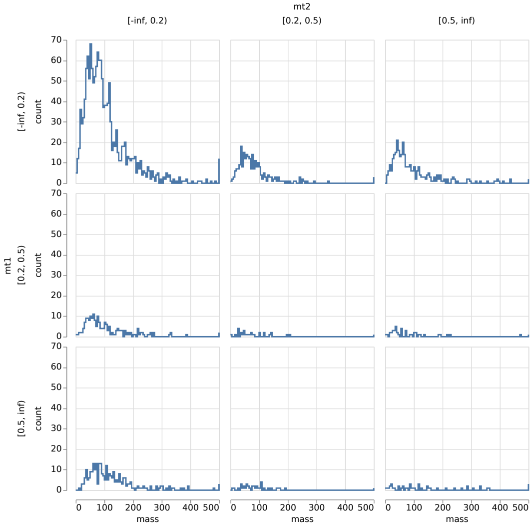
Pandas generalizes what we do with histograms

```
>>> pandhist.steps("mass").column("q1*q2 < 0").data(muonhist)
```



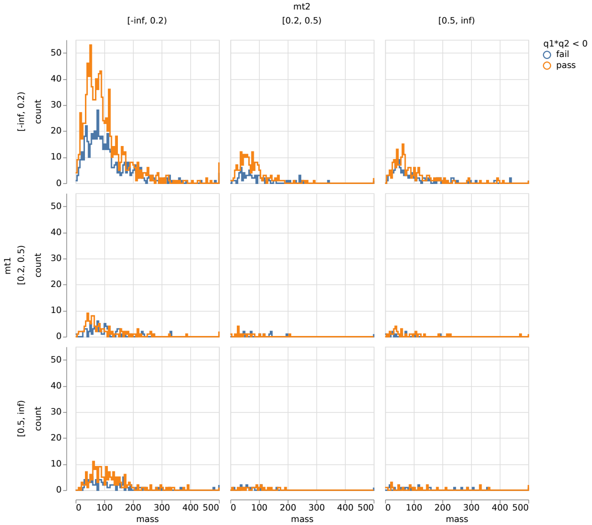
Pandas generalizes what we do with histograms

```
>>> pandhist.steps("mass").row("mt1").column("mt2").data(muonhist)
```

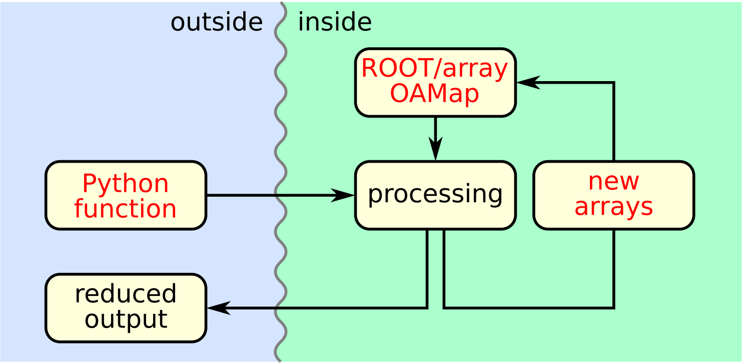


Pandas generalizes what we do with histograms

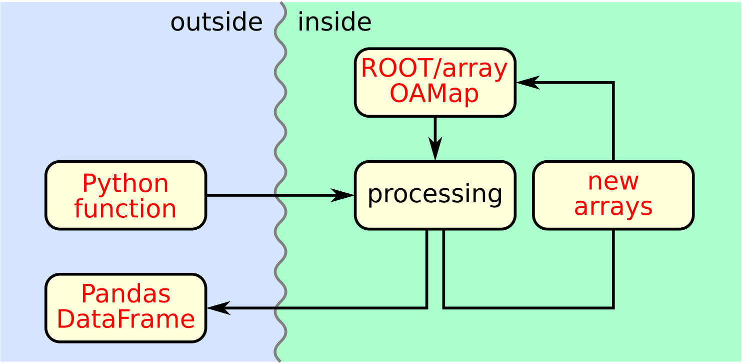
```
>>> pandhist.steps("mass").overlay("q1*q2 < 0").row("mt1").column("mt2").data(muonhist)
```



Current thinking. . .



Current thinking...



What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

- Spark is the elephant in the room (used to be Hadoop), but it's hard to (efficiently) bridge our C++/Python ecosystem with the JVM.

What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

- Spark is the elephant in the room (used to be Hadoop), but it's hard to (efficiently) bridge our C++/Python ecosystem with the JVM.
- I considered Drill, but it's also JVM.

What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

- Spark is the elephant in the room (used to be Hadoop), but it's hard to (efficiently) bridge our C++/Python ecosystem with the JVM.
- I considered Drill, but it's also JVM.
- Impala is C++, but seems to be too specialized to the SQL mindset.

What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

- Spark is the elephant in the room (used to be Hadoop), but it's hard to (efficiently) bridge our C++/Python ecosystem with the JVM.
- I considered Drill, but it's also JVM.
- Impala is C++, but seems to be too specialized to the SQL mindset.
- Thanat Jatuphattharachat (summer student) investigated raw Zookeeper coordination, but this is getting close to DIY.

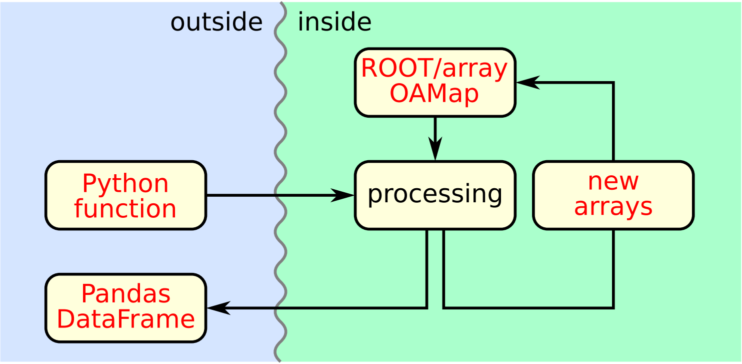
What should our processing be?

Speedy, prompt start-up parallel processing is not something HEP needs to invent.

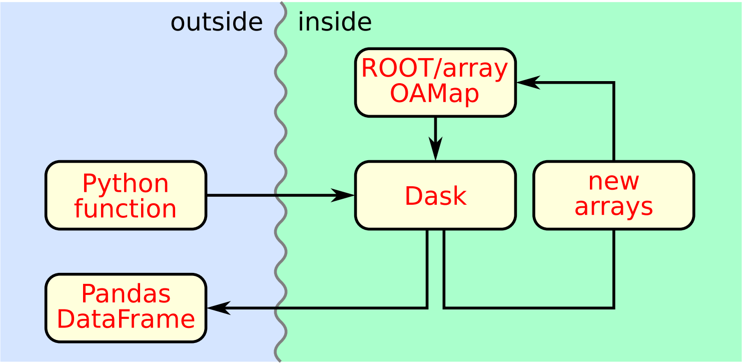
- Spark is the elephant in the room (used to be Hadoop), but it's hard to (efficiently) bridge our C++/Python ecosystem with the JVM.
- I considered Drill, but it's also JVM.
- Impala is C++, but seems to be too specialized to the SQL mindset.
- Thanat Jatuphattharachat (summer student) investigated raw Zookeeper coordination, but this is getting close to DIY.
- Dask looks like a good choice: it's in the C++/Python world, general enough to piece together what we need out of basic parts.

  (More about this in the concurrency session.)

Current thinking...



Current thinking. . .



What should our data storage be?

Again, no need for HEP to invent.

What should our data storage be?

Again, no need for HEP to invent.

- Using columns, rather than files, as the fundamental unit means keeping track of a much larger number of named entities.

What should our data storage be?

Again, no need for HEP to invent.

- Using columns, rather than files, as the fundamental unit means keeping track of a much larger number of named entities.
- Object stores scale to larger namespaces than filesystems by providing fewer operations (listing, directory structure), which we don't need for this project.

What should our data storage be?

Again, no need for HEP to invent.

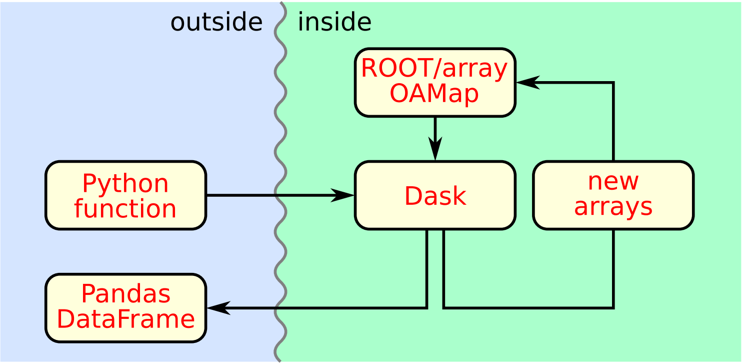
- Using columns, rather than files, as the fundamental unit means keeping track of a much larger number of named entities.
- Object stores scale to larger namespaces than filesystems by providing fewer operations (listing, directory structure), which we don't need for this project.
- Ceph is both an object store and a filesystem, and it minimizes metadata overhead by coordinating data placement through a static function, rather than a dynamic service.

What should our data storage be?

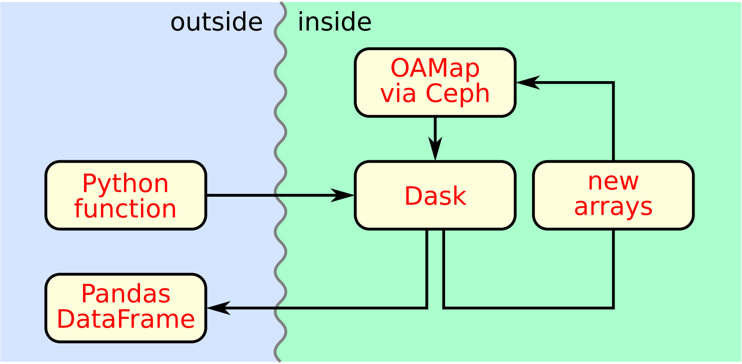
Again, no need for HEP to invent.

- Using columns, rather than files, as the fundamental unit means keeping track of a much larger number of named entities.
- Object stores scale to larger namespaces than filesystems by providing fewer operations (listing, directory structure), which we don't need for this project.
- Ceph is both an object store and a filesystem, and it minimizes metadata overhead by coordinating data placement through a static function, rather than a dynamic service.
  - Lets us experiment both ways.
  - We could use the static function to place executable tasks, improving data locality. . .

Current thinking. . .



Current thinking. . .



Conclusions

Though an analysis service should use as many open source parts as possible, some functionality is missing and we need to build our own.

Software products developed along the way:

uproot	Quickly access ROOT branches as arrays.	mature, in use
oamap	Translate between object-oriented Python and low-level array operations.	ready for testing
pandhist	Reinterpretation of Pandas DataFrames as super-histograms. (Plotting in Vega-Lite.)	experimental
vegascope	Browser-based TCanvas for Vega/Vega-Lite, so that you don't have to use Jupyter if you don't want to.	done (simple)
?	OAMap as a collection in Dask.	
?	Column manager (for sharing data among datasets).	
?	All-in-one environment for query-based analysis.	