Fast Access to Columnar, Hierarchical Data via Code Transformation

Jim Pivarski[1], Peter Elmer[1], Brian Bockelman[2], and Zhe Zhang[2]

[1]Princeton University, [2]University of Nebraska at Lincoln – DIANA-HEP

December 13, 2017

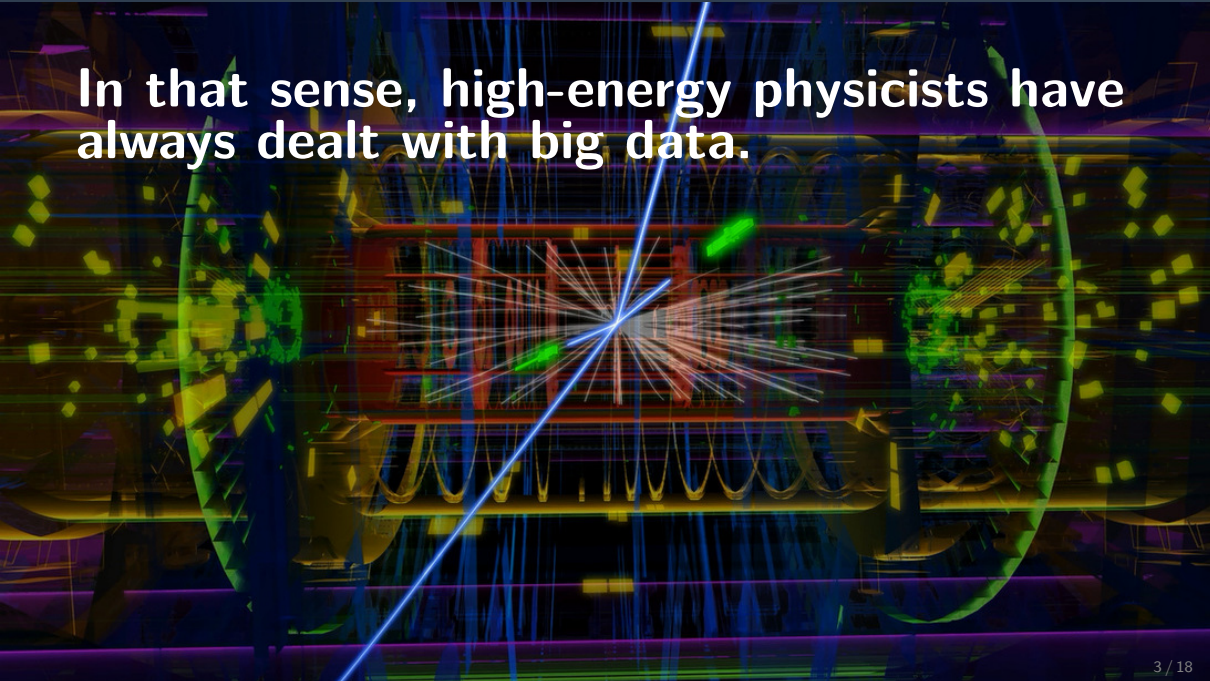
My sliding definition of “big data”

Big data ***(n.)*****:** Techniques for handling datasets that are too big for a single computer.

My sliding definition of "big data"

Big data ***(n.)*****:** Techniques for handling datasets that are too big for a single computer.

This is a moving target: it represents different scales from decade to decade.



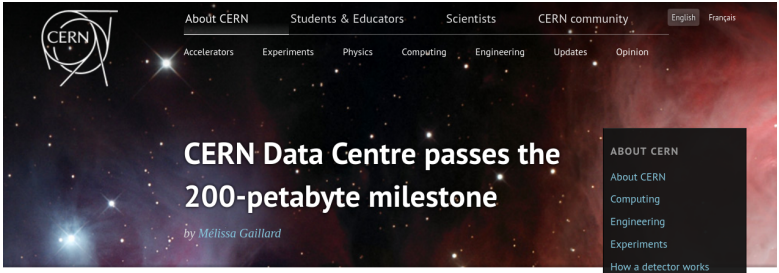
In that sense, high-energy physicists have always dealt with big data.

High-energy physicists dealing with big data

High-energy physicists dealing with big data

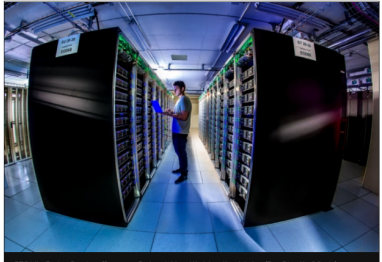
High-energy physicists dealing with big data

But we're certainly not the leaders in big data anymore

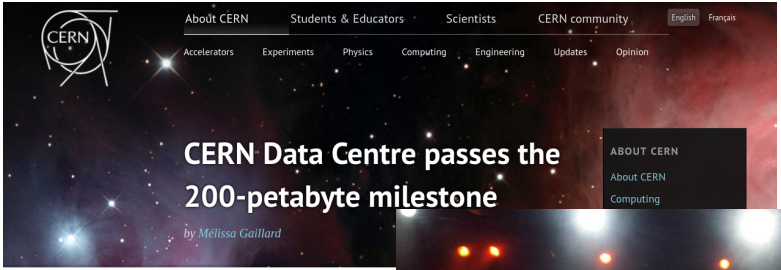


Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.
Voir en français

This content is archived on the CERN Document Server


CERN's Data Centre (Image: Robert Hradil, Monika Majer/ProStudio22.ch)

But we're certainly not the leaders in big data anymore

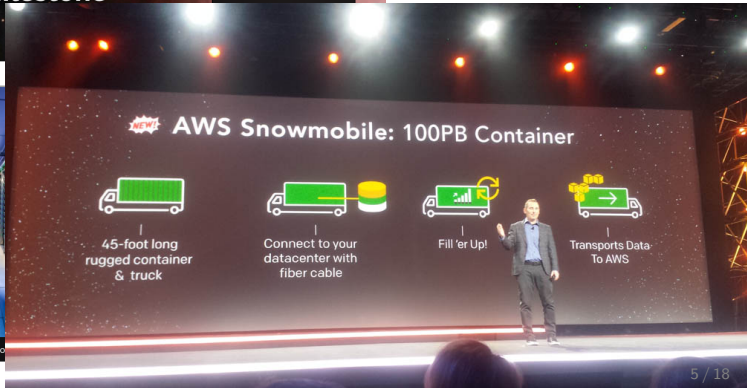


Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.

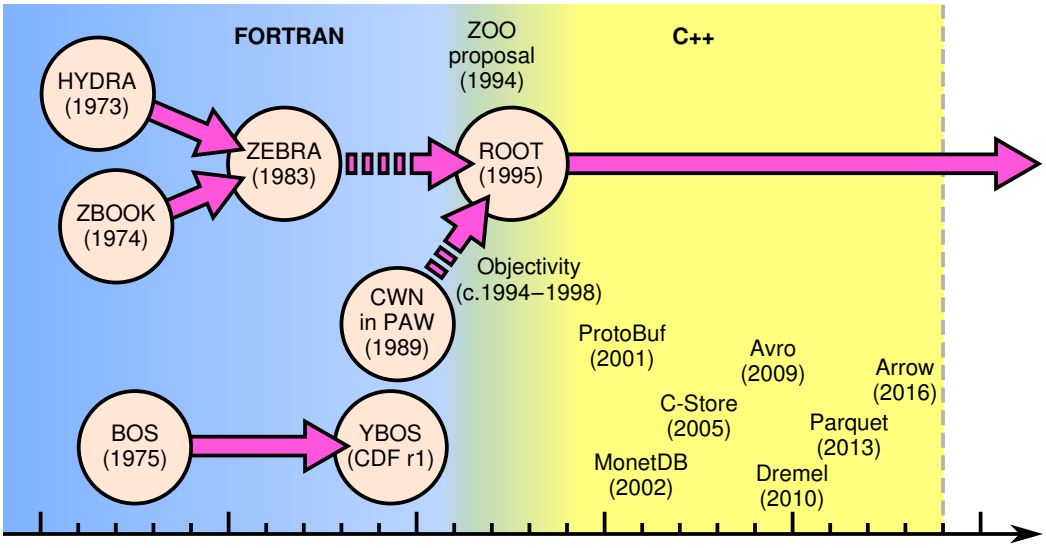
Voir en français

This content is archived on the CERN Document Server

CERN's Data Centre (Image: Robert Hradil, Mo

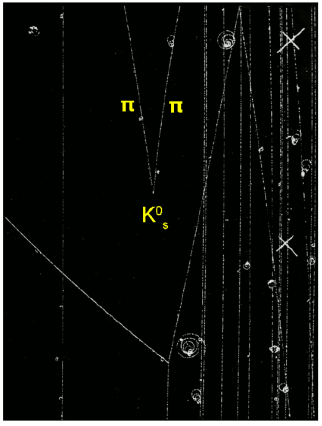


Data structure management packages in high-energy physics



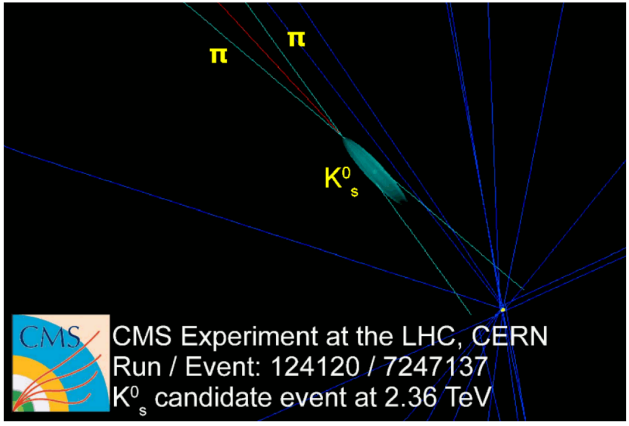
High-energy physics data are arbitrary length lists of objects

Bubble chamber photo



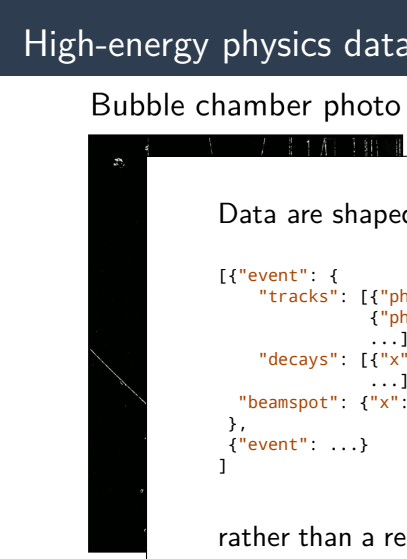
proton beam (upward) on liquid containing protons (stationary)

Reconstructed LHC event

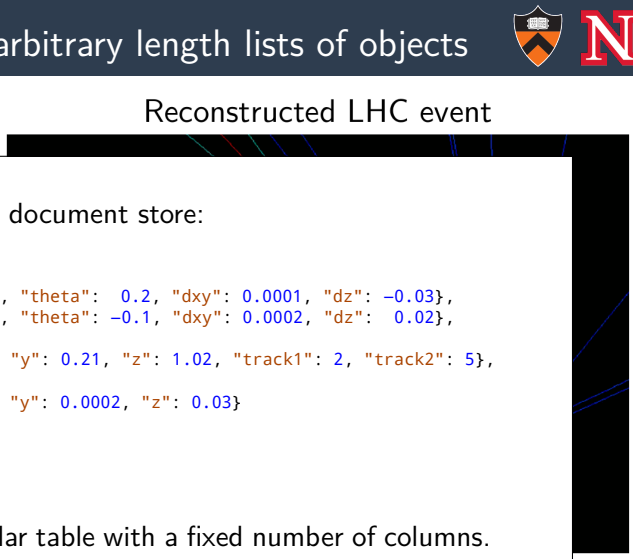


incoming proton beams are perpendicular to this projection, collide at a point and outgoing particles radiate in all directions

High-energy physics data are arbitrary length lists of objects



Bubble chamber photo



Reconstructed LHC event

Data are shaped like a document store:

```
[{"event": {
    "tracks": [{"phi": 3.14, "theta":  0.2, "dxy": 0.0001, "dz": −0.03},
               {"phi": 4.31, "theta": −0.1, "dxy": 0.0002, "dz":  0.02},
               ...],
    "decays": [{"x": −0.12, "y": 0.21, "z": 1.02, "track1": 2, "track2": 5},
               ...],
  "beamspot": {"x": 0.0001, "y": 0.0002, "z": 0.03}
 },
 {"event": ...}
]
```

rather than a rectangular table with a fixed number of columns.

prot his

liquid containing protons (stationary)

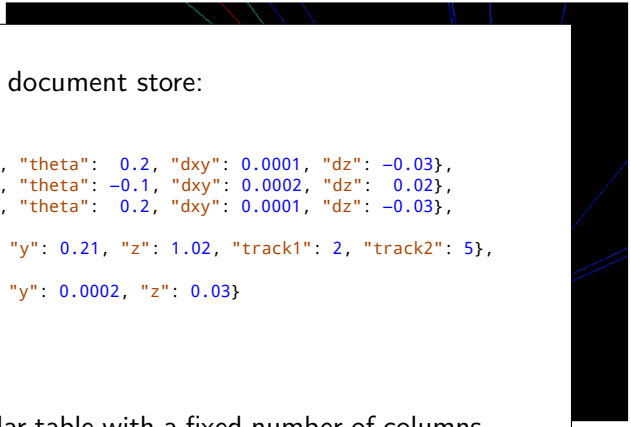
projection, collide at a point and outgoing particles radiate in all directions

High-energy physics data are arbitrary length lists of objects

Bubble chamber photo

(stationary)

Reconstructed LHC event



radiate in all directions

Data are shaped like a document store:

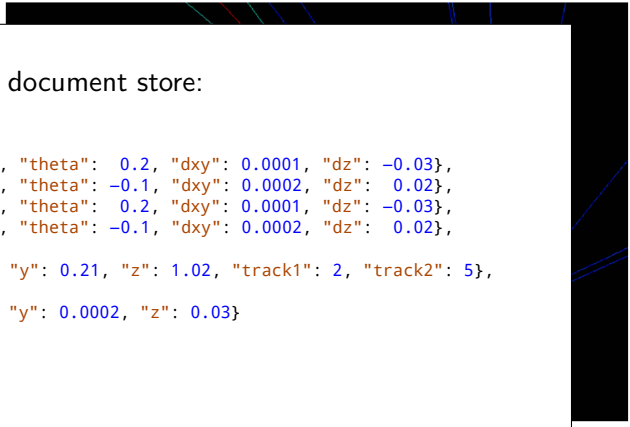
```
[{"event": {
    "tracks": [{"phi": 3.14, "theta":  0.2, "dxy": 0.0001, "dz": −0.03},
               {"phi": 4.31, "theta": −0.1, "dxy": 0.0002, "dz":  0.02},
               {"phi": 3.14, "theta":  0.2, "dxy": 0.0001, "dz": −0.03},
               ...],
    "decays": [{"x": −0.12, "y": 0.21, "z": 1.02, "track1": 2, "track2": 5},
               ...],
  "beamspot": {"x": 0.0001, "y": 0.0002, "z": 0.03}
 },
 {"event": ...}
]
```

rather than a rectangular table with a fixed number of columns.

High-energy physics data are arbitrary length lists of objects

Bubble chamber photo

Reconstructed LHC event



Data are shaped like a document store:

```
[{"event": {
    "tracks": [{"phi": 3.14, "theta":  0.2, "dxy": 0.0001, "dz": −0.03},
               {"phi": 4.31, "theta": −0.1, "dxy": 0.0002, "dz":  0.02},
               {"phi": 3.14, "theta":  0.2, "dxy": 0.0001, "dz": −0.03},
               {"phi": 4.31, "theta": −0.1, "dxy": 0.0002, "dz":  0.02},
               ...],
    "decays": [{"x": −0.12, "y": 0.21, "z": 1.02, "track1": 2, "track2": 5},
               ...],
  "beamspot": {"x": 0.0001, "y": 0.0002, "z": 0.03}
 },
 {"event": ...}
]
```

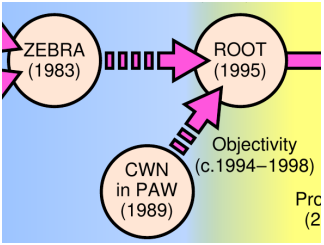
rather than a rectangular table with a fixed number of columns.

prot… liqu… …his …cles

(stationary)

radiate in all directions

We need arbitrary length data structures in a columnar format



The ZEBRA heritage

- Arbitrary length, nested data was a requirement from the very beginning.
- ZEBRA allocated these structures in FORTRAN and saved them to files.
- High energy physics analyses are still batch processes on binary files.

The CWN heritage

- Column-Wise Ntuples: tabular data with same-attribute values stored contiguously in memory/on disk.
- *Much* faster to read just a few attributes, which is essential (particles have $\sim$100 attributes).

Though complex, it's possible to represent arbitrary length, nested data in a columnar format: ROOT (1995), Google Dremel (2010), Apache Parquet (2013), Apache Arrow (2016).

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data: `[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data: `[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`

x attribute `[ a, b, c, d, e, f, g ]`

y attribute `[ 1, 2, 3, 4, 5, 6, 7 ]`

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

```
Logical data: [[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute   [   a,     b,     c,     d,           e,     f,            g     ]
y attribute   [     1,     2,     3,     4,           5,     6,            7   ]
inner stops   [                          4,  4,              6,            7 ]
outer stops   [                                              3,  3,        4]
```

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.
- Random access: `data[0][2][1].x`

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.
- Random access: `data[0][2][1].x` = `x[inner[outer[0] + 2] + 1]` = `f`.

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data: `[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`

x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.
- Random access: `data[0][2][1].x` = `x[inner[outer[0] + 2] + 1]` = `f`.
- This is Arrow format, which is less packed than ROOT, Dremel, or Parquet.

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data: `[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`

x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.
- Random access: `data[0][2][1].x` = `x[inner[outer[0] + 2] + 1] = f`.
- This is Arrow format, which is less packed than ROOT, Dremel, or Parquet.
  - Unified disk and memory formats: ideal for memory mapping or storage-class media.

Example of arbitrary length, nested, columnar data

Type/schema: collection of `List(List(Record(x: char, y: int)))`

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

- Stops array is a cumulative number of items at some level of depth (see how they line up with the closing brackets?).
- More convenient to have both starts and stops ("offsets") by prepending zero.
- Random access: `data[0][2][1].x` = `x[inner[outer[0] + 2] + 1] = f`.
- This is Arrow format, which is less packed than ROOT, Dremel, or Parquet.
  - Unified disk and memory formats: ideal for memory mapping or storage-class media.
  - Can be extremely lightweight: when you access any memory-mapped array element, the operating system reads it and caches on demand. Faster than `fread()`.

Subject of this talk

We want to use zero-copy methods like memory-mapped Arrow to accelerate high-energy physics analysis— enough to enable *interactive*, *exploratory* analysis, rather than just batch processing.

Subject of this talk

We want to use zero-copy methods like memory-mapped Arrow to accelerate high-energy physics analysis— enough to enable *interactive*, *exploratory* analysis, rather than just batch processing.

However, the traditional method of copying serial data into runtime objects like

```
std::vector<std::vector<struct{char x; int y;}>>
```

thwarts the performance advantage of zero-copy data.

Subject of this talk

We want to use zero-copy methods like memory-mapped Arrow to accelerate high-energy physics analysis— enough to enable *interactive*, *exploratory* analysis, rather than just batch processing.

However, the traditional method of copying serial data into runtime objects like

```
std::vector<std::vector<struct{char x; int y;}>>
```

thwarts the performance advantage of zero-copy data.

Instead, we are attempting to transform the user's analysis code to access the data in-place: code transformation as an alternative to deserialization.

Example of code transformation

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner offsets	`[ 0, 4, 4, 6, 7 ]`
outer offsets	`[0, 3, 3, 4]`

Analysis function on logical data

```
for outer in data:
    for inner in outer:
        best = None  # object type
        for record in inner:
            if best is None or \
                    record.y > best.y:
                best = record
        if best is not None:
            print(best.x)
```

Transformed analysis function

```
for i in range(outeroffset[-1]):
    best = -1  # integer type
    for j in range(inneroffset[i],
                   inneroffset[i + 1]):
        if best == -1 or \
                yattr[j] > yattr[best]:
            best = j
    if best != -1:
        print(xattr[best])
```

General features of code transformation

The code transformation obeys rules that can be applied recursively:

- Objects, such as lists and records, become integer-valued indexes.
- List dereferencing (e.g. `data[5]`) becomes offset array dereferencing.
- Offset array indexes for sublists range from `outer[i]` to `outer[i + 1]`, and list length is `(outer[i + 1] - outer[i])`.
- Record dereferencing (e.g. `datum.x`) simply passes the index along to the nested object: `datum.x + datum.y` becomes `xattr[i] + yattr[i]`.
- After transformation, there are no objects, only arrays and indexes.

General features of code transformation

The code transformation obeys rules that can be applied recursively:

- Objects, such as lists and records, become integer-valued indexes.
- List dereferencing (e.g. `data[5]`) becomes offset array dereferencing.
- Offset array indexes for sublists range from `outer[i]` to `outer[i + 1]`, and list length is `(outer[i + 1] - outer[i])`.
- Record dereferencing (e.g. `datum.x`) simply passes the index along to the nested object: `datum.x + datum.y` becomes `xattr[i] + yattr[i]`.
- After transformation, there are no objects, only arrays and indexes.

For our application, we have been transforming a Python abstract syntax tree (AST) and passing the result to Numba (a Python compiler for array-centric code).

Soon, we will attempt to encode these rules in Numba directly as a Numba extension.

Prior art

Apache Drill

- Motivating example: provides fast querying over SQL tables by transforming user queries, rather than materializing row objects.
- However, not for arbitrary length data.

Google Flatbuffers

- Like Google ProtoBufs, except that data are accessed in place, without deserialization.
- However, data are not columnar.

Columnar Objects

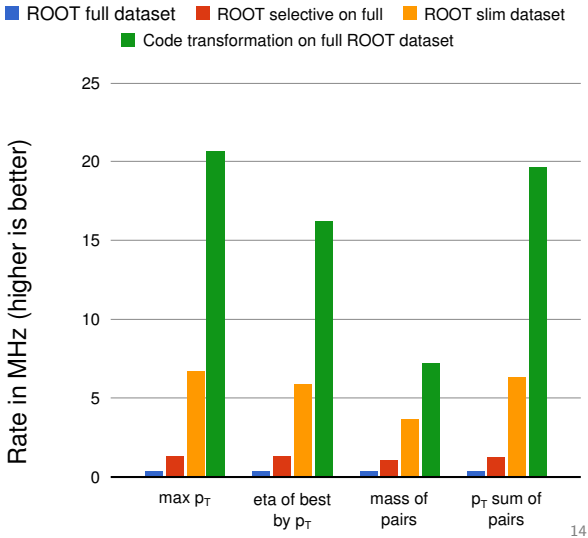
- T. Mattis, J. Henning, P. Rein, R. Hirschfeld, M. Appeltauer, "Columnar Objects: Improving the Performance of Analytical Applications," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015, pp. 197–210.
- Proxy objects in PyPy; similar in that PyPy is JIT-compiled.
- This is the approach we're applying to high-energy physics.

Code transformation outperforms C++ with deserialization

Tested four realistic analysis functions on data from ROOT files with warmed cache.

- ROOT full dataset reconstructs objects with all attributes, including unused ones.
- ROOT selective on full uses dummy placeholders for unused attributes.
- ROOT slim dataset uses exactly what is needed.
- Code transformation on full ROOT dataset is transformed, compiled Python code with no deserialization.

```
def mass_of_pairs(event):
    n = len(event.muons)
    for i in range(n):  # distinct pairs
        for j in range(i+1, n):
            m1 = event.muons[i]
            m2 = event.muons[j]
            mass = sqrt(
                2*m1.pt*m2.pt*(
                cosh(m1.eta - m2.eta) -
                cos(m1.phi - m2.phi)))
            fill_histogram(mass)
```

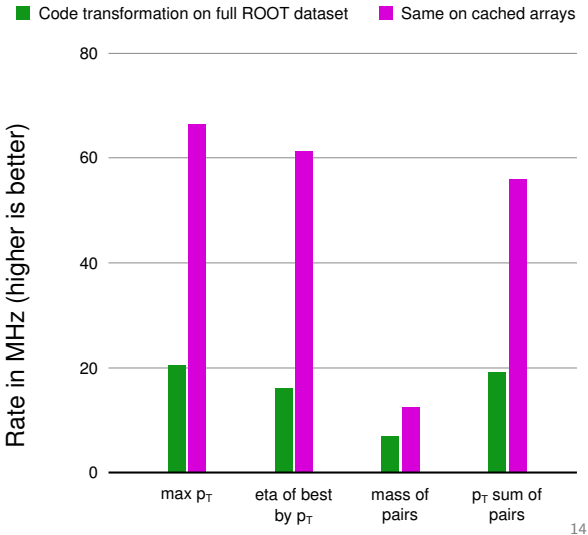


Code transformation outperforms C++ with deserialization

Tested four realistic analysis functions on data from ROOT files or Arrow-like arrays.

- Code transformation on full ROOT dataset same as previous page, for scale.
- Same on cached arrays releases the constraint that data originate in ROOT files, reading them from Arrow-like arrays instead.

```
def mass_of_pairs(event):
    n = len(event.muons)
    for i in range(n):  # distinct pairs
        for j in range(i+1, n):
            m1 = event.muons[i]
            m2 = event.muons[j]
            mass = sqrt(
                2*m1.pt*m2.pt*(
                cosh(m1.eta - m2.eta) -
                cos(m1.phi - m2.phi)))
            fill_histogram(mass)
```



Future work

Although high-energy physics data are non-relational (append-only document stores in columnar representation) exploratory data analysis would be aided by more database-like features.

Future work

Although high-energy physics data are non-relational (append-only document stores in columnar representation) exploratory data analysis would be aided by more database-like features.

- Resilient, distributed storage: we're exploring the option of putting the arrays in an object store like Ceph (one array per key-value pair).

Future work

Although high-energy physics data are non-relational (append-only document stores in columnar representation) exploratory data analysis would be aided by more database-like features.

- Resilient, distributed storage: we're exploring the option of putting the arrays in an object store like Ceph (one array per key-value pair).
- Distributed processing: implementing Hadoop-style locality in the object store (hack Ceph's CRUSH algorithm?).

Future work

Although high-energy physics data are non-relational (append-only document stores in columnar representation) exploratory data analysis would be aided by more database-like features.

- Resilient, distributed storage: we're exploring the option of putting the arrays in an object store like Ceph (one array per key-value pair).
- Distributed processing: implementing Hadoop-style locality in the object store (hack Ceph's CRUSH algorithm?).
- Zero-copy filtering: expanding the data representation (next page) would allow subsets to be expressed without copying list contents, like a database stencil.

Future work

Although high-energy physics data are non-relational (append-only document stores in columnar representation) exploratory data analysis would be aided by more database-like features.

- Resilient, distributed storage: we're exploring the option of putting the arrays in an object store like Ceph (one array per key-value pair).
- Distributed processing: implementing Hadoop-style locality in the object store (hack Ceph's CRUSH algorithm?).
- Zero-copy filtering: expanding the data representation (next page) would allow subsets to be expressed without copying list contents, like a database stencil.
- Database-style indexing: same extension would allow us to sort substructures by a primary key without altering logical order.

Offset array → start, stop arrays

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner offsets	[0, 4, 4, 6, 7]
outer offsets	[0, 3, 3, 4]

Offset array → start, stop arrays

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner starts	[0, 4, 4, 6,]
inner stops	[4, 4, 6, 7]
outer starts	[0, 3, 3,]
outer stops	[3, 3, 4]

Offset array → start, stop arrays for filtering

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner starts	[0, 4, 4, 6,]
inner stops	[4, 4, 6, 7]
outer starts	[0, 3, 3,]
outer stops	[3, 3, 4]
inner starts (v2)	[0, 4]
inner stops (v2)	[1, 5]
outer starts (v2)	[0, 2]
outer stops (v2)	[2, 2]
Logical data (v2):	[[[(a,1)], [(e,5)]], []]

Offset array → start, stop arrays for filtering

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner starts	[0, 4, 4, 6,]
inner stops	[4, 4, 6, 7]
outer starts	[0, 3, 3,]
outer stops	[3, 3, 4]
inner starts (v2)	[0, 4]
inner stops (v2)	[1, 5]
outer starts (v2)	[0, 2]
outer stops (v2)	[2, 2]
Logical data (v2):	[[[(a,1)], [(e,5)]], []]

- inner starts/stops (v2) keeps only the first record of each sublist: particle selection.
- outer starts/stops (v2) keeps only the first two sublists: event selection.

Offset array → start, stop arrays for sorting

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner starts	[0, 4, 4, 6,]
inner stops	[4, 4, 6, 7]
outer starts	[0, 3, 3,]
outer stops	[3, 3, 4]

Offset array → start, stop arrays for sorting

Logical data:	`[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
x attribute	`[ a, b, c, d, e, f, g ]`
y attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`
inner starts	`[ 0, 4, 4, 6, ]`
inner stops	`[ 4, 4, 6, 7 ]`
outer starts	`[0, 3, 3, ]`
outer stops	`[ 3, 3, 4]`
x attribute (v2)	`[ g, e, f, a, b, c, d ]`
y attribute (v2)	`[ 7, 5, 6, 1, 2, 3, 4 ]`
inner starts (v2)	`[ 3, 1, 1, 0 ]`
inner stops (v2)	`[ 7, 1, 3, 1 ]`
Logical data (v2):	unchanged!

Offset array → start, stop arrays for sorting

Logical data:	[[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]
x attribute	[a, b, c, d, e, f, g]
y attribute	[1, 2, 3, 4, 5, 6, 7]
inner starts	[0, 4, 4, 6,]
inner stops	[4, 4, 6, 7]
outer starts	[0, 3, 3,]
outer stops	[3, 3, 4]
x attribute (v2)	[g, e, f, a, b, c, d]
y attribute (v2)	[7, 5, 6, 1, 2, 3, 4]
inner starts (v2)	[3, 1, 1, 0]
inner stops (v2)	[7, 1, 3, 1]
Logical data (v2):	unchanged!

- Physical sort order concentrates the most likely to be requested elements at one end of a large buffer, so that fewer disk pages need to be read/kept in cache.

The big picture

- We're looking for ways to speed up complex, combinatoric user code on columnar data representing arbitrary length lists of objects.
- The Apache Arrow representation unifies memory and storage format, minimizing overhead from disk to RAM to CPU.
- Code transformation, rather than deserialization, further reduces runtime overhead.
- Extension of Arrow's data model enables database-style filtering and sorting.
- If this interests you as a computer scientist researcher or as a data scientist user and you want to collaborate, please contact me!

Jim Pivarski <pivarski@fnal.gov>