Managing data with columnar granularity

Jim Pivarski

Princeton University – DIANA-HEP

November 16, 2017

Nature of this talk

This talk isn't about how we manage data in HEP, but how we *might.*

- Therefore, it isn't a "how-to" talk but a "what-if" talk.
- If you have experience in this, I want to hear from you!

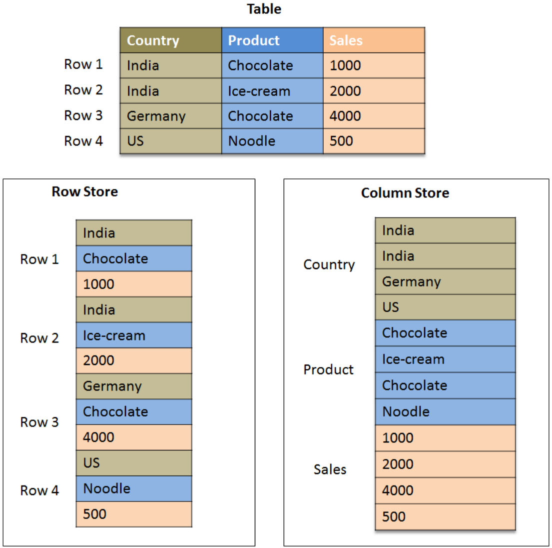
Columnar data

Serializing data in columns is an old idea in HEP:

- 1989: Column-Wise-N-tuples (CWN) in PAW
- 1996: “split” (columnar) C++ objects in ROOT

. . .

- 2002: MonetDB
- 2005: C-Store (Vertica)
- 2010: Google Dremel paper
- 2013: Apache Parquet
- 2016: Apache Arrow



Table

	Country	Product	Sales
Row 1	India	Chocolate	1000
Row 2	India	Ice-cream	2000
Row 3	Germany	Chocolate	4000
Row 4	US	Noodle	500

Row Store

Row 1	India Chocolate 1000
Row 2	India Ice-cream 2000
Row 3	Germany Chocolate 4000
Row 4	US Noodle 500

Column Store

Country	India India Germany US
Product	Chocolate Ice-cream Chocolate Noodle
Sales	1000 2000 4000 500

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[3, 3, 4]`
inner stops	`[4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[3, 3, 4]`
inner stops	`[4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]]`
outer stops	`[3, 3, 4]`
inner stops	`[4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.
- Stops array: cumulative number of items for some level at each *closing* bracket.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[ 3, 3, 4]`
inner stops	`[ 4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.
- Stops array: cumulative number of items for some level at each *closing* bracket.
- Alternative representations:
    - Offsets (Arrow): include *starting* index; can represent interval slices without copying.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[3, 3, 4]`
inner stops	`[4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.
- Stops array: cumulative number of items for some level at each *closing* bracket.
- Alternative representations:
  - Offsets (Arrow): include *starting* index; can represent interval slices without copying.
  - Starts and stops: `starts, stops = offsets[:-1], offsets[1:]`; can represent union of interval slices without copying, even save out of order for indexed lookups.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[3, 3, 4]`
inner stops	`[4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.
- Stops array: cumulative number of items for some level at each *closing* bracket.
- Alternative representations:
  - Offsets (Arrow): include *starting* index; can represent interval slices without copying.
  - Starts and stops: `starts, stops = offsets[:-1], offsets[1:]`; can represent union of interval slices without copying, even save out of order for indexed lookups.
  - Sizes: `sizes = stops - starts`; compressible, fill in parallel, but no $\mathcal{O}(1)$ lookup.

Hierarchically nested columnar data

Rowwise → columnar is a transposition for tabular data; nested data is more complex.

Example: `vector<vector<pair<char, int>>>`

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[ 3, 3, 4]`
inner stops	`[ 4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

- Each primitive attribute is in an array by itself, with no list boundaries.
- Stops array: cumulative number of items for some level at each *closing* bracket.
- Alternative representations:
  - Offsets (Arrow): include *starting* index; can represent interval slices without copying.
  - Starts and stops: `starts, stops = offsets[:-1], offsets[1:]`; can represent union of interval slices without copying, even save out of order for indexed lookups.
  - Sizes: `sizes = stops - starts`; compressible, fill in parallel, but no $\mathcal{O}(1)$ lookup.
  - Dremel/Parquet: "repetition level"; packed small integers, but no $\mathcal{O}(1)$ lookup.

Although we know how to save and retrieve data in columnar form, we still manage data *as files.*

Although we know how to save and retrieve data in columnar form, we still manage data *as files.*

Whether it's ROOT or Parquet, the file structure glues a set of columns together to be downloaded, replicated, versioned, or migrated to colder storage *as a unit.*

Why is this a problem?

The reason columnar data is so useful is because each end-user analysis requires a minority of the data columns.

Why is this a problem?

The reason columnar data is so useful is because each end-user analysis requires a minority of the data columns.

- “Monojet analysis” only needs jet objects, but it needs jets constructed many different ways to study systematics.
- “Boosted top search” needs jets with substructure variables.
- “Heavy flavor study” needs jets, electrons, and muons with isolation and B-tagging variables.
- “Diphoton Higgs mass” needs photons, electrons for a veto, and converted pair electrons.
- . . .

Why is this a problem?

The reason columnar data is so useful is because each end-user analysis requires a minority of the data columns.

- “Monojet analysis” only needs jet objects, but it needs jets constructed many different ways to study systematics.
- “Boosted top search” needs jets with substructure variables.
- “Heavy flavor study” needs jets, electrons, and muons with isolation and B-tagging variables.
- “Diphoton Higgs mass” needs photons, electrons for a veto, and converted pair electrons.
- . . .

Within each particle object, the kinematic variables (p_T, η, ϕ, m) are needed the most, with “isolation/tagging/matching/. . .” needed by different analyses to varying degrees.

Why is this a problem?

The reason columnar data is so useful is because each end-user analysis requires a minority of the data columns.

- “Monojet analysis” only needs jet objects, but it needs jets constructed many different ways to study systematics.

  converted pair electrons.

- . . .

Columnar data lets us read relevant attributes from disk one at a time (or with XRootD, over the network), but data management systems are unaware of how to open up a ROOT file and operate on individual columns.

Within each particle object, the kinematic variables (p_T, η, ϕ, m) are needed the most, with “isolation/tagging/matching/. . .” needed by different analyses to varying degrees.

What could we do if data management were column-aware?

Case 1: serve the most desirable attributes from RAM or SSD and less desirable attributes of the same dataset from disk or tape.

Currently, we make 2 or 3 levels of “slimmed” copies (AOD/MiniAOD/NanoAOD) to serve with different latencies. Three sizes does not fit all, so individual analysis groups make their own subsets (and have to find their own storage).

What could we do if data management were column-aware?

Case 1: serve the most desirable attributes from RAM or SSD and less desirable attributes of the same dataset from disk or tape.

Currently, we make 2 or 3 levels of "slimmed" copies (AOD/MiniAOD/NanoAOD) to serve with different latencies. Three sizes does not fit all, so individual analysis groups make their own subsets (and have to find their own storage).

Case 2: define datasets with overlapping sets of physical columns.

For instance, version 1 has incorrect jet energy corrections; version 2 is just like it but with different jet energies. Versions 1 and 2 should share the same physical storage for all other columns. (Currently, users pass around *correction recipes!*)

What could we do if data management were column-aware?

Case 1: serve the most desirable attributes from RAM or SSD and less desirable attributes of the same dataset from disk or tape.

Currently, we make 2 or 3 levels of “slimmed” copies (AOD/MiniAOD/NanoAOD) to serve with different latencies. Three sizes does not fit all, so individual analysis groups make their own subsets (and have to find their own storage).

Case 2: define datasets with overlapping sets of physical columns.

For instance, version 1 has incorrect jet energy corrections; version 2 is just like it but with different jet energies. Versions 1 and 2 should share the same physical storage for all other columns. (Currently, users pass around *correction recipes!*)

Case 3: provide zero-copy views of selected particles or events through stencils/bitmaps.

Currently, users make “skimmed” copies, which use more space and can’t benefit from version updates such as the jet energy correction example above.

What could we do if data management were column-aware?

Case 1: serve the most desirable attributes from RAM or SSD and less desirable attributes of the same dataset from disk or tape.

Currently, we make 2 or 3 levels of "slimmed" copies (AOD/MiniAOD/NanoAOD) to serve with different latencies. Three sizes does not fit all, so individual analysis groups make their own subsets (and have to find their own storage).

Case 2: define datasets with overlapping sets of physical columns.

For instance, version 1 has incorrect jet energy corrections; version 2 is just like it but with different jet energies. Versions 1 and 2 should share the same physical storage for all other columns. (Currently, users pass around *correction recipes!*)

Case 3: provide zero-copy views of selected particles or events through stencils/bitmaps.

Currently, users make "skimmed" copies, which use more space and can't benefit from version updates such as the jet energy correction example above.

Case 4: speed up filtering with database-style indexing.

Case 1: lower latency for popular columns

Same object-array mapping example:

logical data	`[[`	`(a,1),`	`(b,2),`	`(c,3),`	`(d,4)],`	`[],`	`[(e,5),`	`(f,6)]],`	`[],`	`[[(g,7)]`	`]`
outer stops	`[`							`3,`	`3,`		`4]`
inner stops	`[`				`4,`	`4,`		`6,`		`7`	`]`
1st attribute	`[`	`a,`	`b,`	`c,`	`d,`		`e,`	`f,`		`g`	`]`
2nd attribute	`[`	`1,`	`2,`	`3,`	`4,`		`5,`	`6,`		`7`	`]`

If the 2nd attribute is more popular than the 1st attribute, raise the 2nd attribute into warmer cache (on the server).

To the degree that analysts' interests overlap (e.g. the all-popular kinematic variables), one copy in hot cache may be shared by all. This is impossible for private skims.

Case 2: overlapping dataset definitions

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[ 3, 3, 4]`
inner stops	`[ 4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute (v1)	`[ 1, 2, 3, 4, 5, 6, 7 ]`
2nd attribute (v2)	`[ 9, 9, 9, 9, 9, 9, 9 ]`

Dataset version 1 schema:

```
List(stops = "outer stops",
  List(stops = "inner stops",
    Pair(first = "1st attribute",
         second = "2nd attribute (v1)"
    )))
```

Dataset version 2 schema:

```
List(stops = "outer stops",
  List(stops = "inner stops",
    Pair(first = "1st attribute",
         second = "2nd attribute (v2)"
    )))
```

(Not all arrays can be combined into datasets; validity determined by provenance.)

Case 3: zero-copy views of selections

logical data	[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]
outer offsets	[0, 3, 3, 4]
inner offsets	[0, 4, 4, 6, 7]
1^{st} attribute	[a, b, c, d, e, f, g]
2^{nd} attribute	[1, 2, 3, 4, 5, 6, 7]
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 pair of each sublist: particle selection.
- outer starts/stops (v2) keeps only the first two sublists: event selection.

Case 3: zero-copy views of selections

logical data	[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]
outer offsets	[0, 3, 3, 4]
inner offsets	[0, 4, 4, 6, 7]
1^{st} attribute	[a, b, c, d, e, f, g]
2^{nd} attribute	[1, 2, 3, 4, 5, 6, 7]
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 pair of each sublist: particle selection.
- outer starts/stops (v2) keeps only the first two sublists: event selection.
- If a new 2^{nd} attribute is created, we can immediately update the selected data.

Case 4: database-style indexing (only a sorting example)

logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer offsets	`[0, 3, 3, 4]`
inner starts	`[0, 4, 4, 6]`
inner stops	`[4, 4, 6, 7]`
1st attribute	`[a, b, c, d, e, f, g]`
2nd attribute	`[1, 2, 3, 4, 5, 6, 7]`
1st attribute (v2)	`[g, e, f, a, b, c, d]`
2nd 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!

Case 4: database-style indexing (only a sorting example)

```
      logical data  [[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]
     outer offsets  [0,                                              3,  3,         4]
      inner starts  [0,                            4,  4,                    6       ]
       inner stops  [4,                            4,  6,                    7       ]
     1st attribute  [  a,     b,     c,     d,           e,     f,             g     ]
     2nd attribute  [    1,     2,     3,     4,           5,     6,             7   ]
1st attribute (v2)  [  g,           e,     f,            a,     b,     c,     d      ]
2nd 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!
```

- Different sublists can be sorted differently, e.g. muon attributes by max muon p_T per event and jet attributes by max jet p_T per event.

Case 4: database-style indexing (only a sorting example)

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

     outer offsets  [0,                                              3,  3,        4]
      inner starts  [0,                            4,  4,                    6      ]
       inner stops  [4,                            4,  6,                    7      ]
     1st attribute  [  a,     b,     c,     d,           e,     f,             g    ]
     2nd attribute  [    1,     2,     3,     4,           5,     6,             7  ]

1st attribute (v2)  [  g,           e,     f,            a,     b,     c,     d     ]
2nd 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!
```

- Different sublists can be sorted differently, e.g. muon attributes by max muon p_T per event and jet attributes by max jet p_T per event.
- Request for ${p_T}^{\text{muon}} > X$ AND ${p_T}^{\text{jet}} > Y$ only touches one end of all the arrays.

How might it be implemented?

This is not an Object-Relational Mapping (ORM): the order of the arrays is important and they should be served in contiguous blocks.

→ suggests array database (e.g. SciDB) or object store (e.g. Ceph)

How might it be implemented?

This is not an Object-Relational Mapping (ORM): the order of the arrays is important and they should be served in contiguous blocks.

→ suggests array database (e.g. SciDB) or object store (e.g. Ceph)

Option 1: define an Object-Array Mapping (OAM), build an interpretive layer around an object store, and translate HEP data into it.

Option 2: use ROOT's OAM and interpretive layer, but replace its file-backed storage with the object store.

How might it be implemented?

This is not an Object-Relational Mapping (ORM): the order of the arrays is important and they should be served in contiguous blocks.

→ suggests array database (e.g. SciDB) or object store (e.g. Ceph)

Option 1: define an Object-Array Mapping (OAM), build an interpretive layer around an object store, and translate HEP data into it.

Option 2: use ROOT's OAM and interpretive layer, but replace its file-backed storage with the object store.

Option 2 is more limited (no start/stop arrays), but less needs to be invented and old analysis scripts would function in the new system.

ROOT I/O → ROOT object store

ROOT I/O

- File accessed by a single user contains objects and subdirectories.

ROOT object store

- A server-bound file view would have many users, “home directories.”

ROOT I/O → ROOT object store

ROOT I/O

- File accessed by a single user contains objects and subdirectories.

ROOT object store

- A server-bound file view would have many users, “home directories.”
- Adopt object store’s security model.

ROOT I/O → ROOT object store

ROOT I/O

- File accessed by a single user contains objects and subdirectories.
- Segments of columnar arrays called "baskets" are located in the file, identified by file seek positions.

ROOT object store

- A server-bound file view would have many users, "home directories."
- Adopt object store's security model.
- Same baskets would be identified by object store keys. No fragmentation concerns and objects get replicated.

ROOT I/O → ROOT object store

ROOT I/O

- File accessed by a single user contains objects and subdirectories.
- Segments of columnar arrays called "baskets" are located in the file, identified by file seek positions.
- Users typically access a large number of identically typed files.

ROOT object store

- A server-bound file view would have many users, "home directories."
- Adopt object store's security model.
- Same baskets would be identified by object store keys. No fragmentation concerns and objects get replicated.
- No artificial boundaries in the dataset: only segmented into baskets, which are hidden from users.

ROOT I/O → ROOT object store

ROOT I/O

- File accessed by a single user contains objects and subdirectories.
- Segments of columnar arrays called "baskets" are located in the file, identified by file seek positions.
- Users typically access a large number of identically typed files.

ROOT object store

- A server-bound file view would have many users, "home directories."
- Adopt object store's security model.
- Same baskets would be identified by object store keys. No fragmentation concerns and objects get replicated.
- No artificial boundaries in the dataset: only segmented into baskets, which are hidden from users.
- Need to develop new interfaces to share basket data among versioned datasets and track provenance.

Questions for HEP: any use-case concerns? Missing features?

Questions for others: does this look familiar? Do you have experience with systems like this? If so, what worked/didn't work?