HSF Community White Paper Contribution

Hierarchical Data with Columnar Granularity

Tanu Malik[a], **Mark Neubauer**[b], **Jim Pivarski**[c], **Peter Elmer**[c]

[a]*DePaul University, Chicago, Illinois, USA*
[b]*University of Illinois at Urbana-Champaign, Urbana, Illinois, USA*
[c]*Princeton University, Princeton, New Jersey, USA*

Introduction

Although most High Energy Physics (HEP) datasets are converted into tabular structures ("ntuples" or "data summary tables") in the final stages of analysis, the general-use Analysis Object Datasets (AOD) prepared centrally for the hundreds or thousands of members of each collaboration must have a hierarchical structure[1]. HEP events (snapshots of a collision or fixed-target interaction) *contain* particle tracks, calorimeter deposits, and other signals, which are reconstructed as arbitrarily-many particle *objects* with *attributes* such as momentum, charge, isolation with respect to other particles, etc. Containment and cross-linking are the most natural ways to think about these data.

Traditionally, these data have been stored as C++ objects, serialized by the ROOT framework into ROOT files. The features of this format were implemented in response to physics needs: from flat tables to user-customizable object serializations to standardized object serializations for most C++ types, and from row-wise representations to columnar representations. In this document, we refer to objects of homogeneous type (such as HEP events) as the "rows" of the data and their named, typed attributes as "columns." A "row-wise" layout stores whole objects contiguously, with all attributes of a given object in close proximity within the file or byte stream, while a "columnar" layout stores all values of

[1] We reserve the word "hierarchical" for nested data structures with arbitrary-length lists. Nesting of fixed-size objects, such as the four components of particle momentum, is merely a semantical convenience, not a structural necessity.

a given attribute contiguously, such that a single object is split across the file or byte stream.

This columnar layout is a crucial feature of the ROOT file format for HEP data. Typical data analysis procedures access only a few types of particles from each event and only a few attributes from each particle object. A columnar layout minimizes the number of disk pages that need to be touched to fetch the data. Furthermore, splitting data into columns allows for each attribute to be separately compressed, which leads to higher compression ratios because a single attribute's distribution has lower entropy than all the attributes of a complex object. Columnar, hierarchical data became a staple of HEP datasets years before Google's famous Dremel paper (independently) introduced this concept to the big data world, as well as the Parquet file format that was based on Dremel.

However, ROOT serialization (and the HEP community's use of it) has some limitations. Its object serialization is bound to the C++ type system, making it unclear how to translate certain objects into Python frameworks or big data tools like Spark. The C++ type system is designed for runtime objects and is unnecessarily complex for describing static data. In an effort to optimize storage, HEP collaborations have customized the serialization of some of their data types, which reintroduces the problem that ROOT serialization was intended to solve: custom software is needed to read the data. Custom software exacerbates the problem of data preservation.

Binding the serialized data's schemas to C++ types created another problem: versioning those data types. One of ROOT's strengths is that it has a built-in system for versioning class definitions and gracefully filling new classes with old data (known as "schema evolution"). However, this wouldn't even be a problem if objects were not bound to C++ classes.

Being an evolving format with backward compatibility, ROOT serialization has some incidental limitations and inefficiencies. Variable-length structures cannot be split more than one level deep. Counters for variable-length structures are duplicated for every attribute of a class. Class version numbers are repeated for all objects in a column, despite the fact that they cannot change within the column. Columns are partitioned into pages ("baskets") that aren't required to line up at regular intervals ("clusters"), though read-out is more efficient when they do. Each of these could be improved by incremental updates and, in some cases, forward compatibility breaks.

However, a more basic issue cannot be addressed without a fundamentally new method of data management: each self-contained file binds a fixed set of particles and attributes together as a granular unit. ROOT's internally columnar layout is a step toward breaking this unit, in that analysis functions can selectively read the columns they want, even over a network (XRootD), but storing data and persistently caching it at local sites are still performed on a per-file basis, taking all columns within the file, regardless of whether they are all needed.

For this reason, HEP collaborations and analysis groups go to great lengths to minimize the number of bytes per event in their data structures ("data tiers"), often making sacrifices to do so. For instance, CMS's MiniAOD, NanoAOD, informal Bacon, Panda, Cms3, TreeMaker, and ATLAS's derived DxAODs all represent different selections of columns that can be used by different subsets of the collaboration. Often, a critical quantity for one analysis is excluded from a data tier so that the resulting files are small enough to be convenient for other analyses. Since ROOT files are self-contained, each selection of columns ("slimming") and selection of events ("skimming") must be a separate copy on disk. The smaller the files are, the more one needs.

In this document, we describe a new method of data management in which a partitioned column of data is the fundamental unit. We describe a simplified splitting/serialization procedure within an abstract type system, accessible to any language or data framework, which can be applied to any hierarchical data. These data may represent HEP events or auxiliary data, such as lookup tables, calibration constants, or machine learning models, some of which aren't even lists at their topmost level. We also show how this system can be used within a centralized service to slim and skim data without any copying, and can be projected for specific analysis needs without multiplying data formats. Finally, we show how this data representation provides two new features, never before available to the HEP community but known to increase the efficiency of databases: zero-deserialization scanning and database-style indexing.

Columnar granularity

What we are advocating is to present column partitions as the fundamental unit of data management for HEP end-user analysis (not reconstruction). These column partitions would become first-class citizens in the same sense that files are today: either as single-column files or more likely as binary blobs in an object store. Analysis software would retrieve columns individually, so that a single collection of columns could suit the needs of all analysis groups. Columns that are less frequently accessed may regress to colder storage, but they wouldn't have to be selected upfront by data-tier designers.

In ROOT terminology, column partitions are called "baskets," which represent one attribute for an integer number of events ("entries"). All of the columns in a partition are called a "cluster." ROOT baskets are typically kilobytes; we would prefer merging all baskets in a file or several files to make megabyte-sized column partitions. Megabytes per column partition is large enough to optimize high-throughput loops and small enough to manage thousands in memory at a time. ROOT has a mechanism for using data from different datasets ("trees") as though they were parts of the same dataset ("tree friends"); this would be a similar idea, applied universally.

Given a data store full of columns, datasets become loose associations among these columns, metadata identifying a set of columns as mutually consistent and meaningful for analysis. To share columns among datasets, they must be immutable, but HEP datasets have traditionally been viewed as immutable, anyway. This is an especially powerful mechanism for versioning a dataset, since the "replacement" of a column of data with updated data is now extremely lightweight: only the new column needs to be additionally stored, not a full copy. Slimming a dataset (removing columns) becomes a metadata-only operation and would be performed implicitly by executing code that touches a subset of particle attributes.

If all data are accessed on a centralized server, skims (removing entries) can be "soft." The result of a complicated filter can be stored as a list of selected entry indexes, and subsequent operations on the filtered data go through the entry index. In HEP, this technique is known as an "event list." A soft skim reuses data and is therefore very lightweight, but has the disadvantage that the original (large) dataset must be available.

This is feasible on a shared server, but not for a local copy. To download a copy or repartition the reduced data, a traditional “hard skim” would need to be generated.

We should point out that this proposal goes beyond what is done in the big data industry—Parquet, Apache Arrow, and SparkSQL all store data as columns but bind those columns in files (Parquet’s case) or data structures (Arrow and SparkSQL) that are treated as units. We believe that the need for data with columnar granularity is driven by HEP, which deals with data that are more complex and more nested than most in the big data industry. It may someday find application in industry as well.

Splitting hierarchical data into columns

It’s easy to see how a flat table can be converted from a row-wise to a columnar format: simply transpose the table in memory. The same is true for data structures containing fixed-size nested records, such as “particle” containing “energy” and “mass.” Replace the record types with primitives that have composite names, such as “particle.energy” and “particle.mass”, reducing the data structure to a flat table.

For hierarchical data with arbitrary-length lists, it is less clear. If each “event” row has arbitrarily many “particles,” the “energy” and “mass” columns have a different number of elements for each row. We could store all “particle.energy” values in a columnar array, followed by all “particle.mass” values, but then we have lost the boundaries that separate one event from the next. We need an additional “#particles” column, but this column has a different length than “particle.energy” and “particle.mass”, as there is exactly one “#particles” per event.

Columnar splitting techniques

There are at least five fundamentally different methods to split hierarchical data into columns (also known as “shredding”).

The first of these, described above, flattens the attribute data and adds a counter that can reconstruct the original structure. It is the method used by ROOT, though only for one level of depth. It could be extended to allow lists of lists, or more likely, lists of records whose fields are lists, by introducing a separate counter column for each level of depth.

This method cannot handle recursively defined types, such as a Tree type that contains a list of Tree children, because arbitrarily deep structures would require an unknown (until the data are observed) number of columns. It is also not suited for random access, since the index of the first particle in event 1000 can only be found by adding up all the "#particles" counts in the first 99 events.

The second method, used by Apache Arrow and SparkSQL, replaces the counters with offsets. The offsets are simply cumulative sums of counts, but they permit random access: an item with coordinates `(i, j, k)` can be found via offset columns `outer`, `middle`, and `inner` as:

```
inner[middle[outer[i] + j] + k]
```

However, it is still not possible to represent recursively defined types.

The third method, which roughly corresponds to normal form in SQL, adds an "event id" to the particle columns instead of adding a counter or offset to the events. Thus, all particles belonging to the first event have "event id" with value 1, all particles belonging to the second event have "event id" 2, etc. This tag is an attribute of the particle, just like energy and mass, so the particle data forms a flat, rectangular table, and the particle table has no intrinsic ordering (good for SQL, bad for random access).

This is particularly advantageous if the number of particles per event is so large that they must be distributed among independent processors. Aggregating one quantity per event becomes an SQL group by operation, which in principle examines the entire dataset for matching event identifiers. HEP data occupy a completely different regime, though— the number of particles per event is small enough that they easily fit in one processing engine, and so it would be wasteful to examine the whole dataset or even a partition to find particles belonging to a given event. It is much more efficient to keep particles belonging to a given event contiguous, find them by index, and ensure that they are never split between independent processors.

A fourth method, which we have never seen in practice, addresses the problem of recursively defined types. Nested lists of lists do not need to be represented by separate counter or offset columns: the counters for all levels of nesting can be rolled up into one column. We call this method "recursive counters."

Consider the following example:

```
[ [(1, 1.1)], [], [(2, 2.2), (3, 3.3)] ], [ [(4, 4.4)] ]
```

The square brackets represent lists and the parentheses represent a record with two fields, "a" and "b." The list brackets are colored to indicate depth of structure.

The structure of both list levels can be packed into a single counter as follows (with flattened attribute data in separate columns):

```
counter 3,1,         0,  2,                         1,1
data-a      1,               2,        3,              4
data-b         1.1,             2.2,      3.3,            4.4
```

The first value in the counter gives the length of an outer list, which is 3, and the next three values (1, 0, 2) give the lengths of inner lists. The next value after that gives the length of the next outer list, which is 1, followed by one value for the length of its inner list (1). Starting at the beginning, one can always interpret whether a value pertains to an outer list or an inner list by maintaining a stack of counter indexes. There is no maximum depth of nesting if there is no maximum depth for this stack of counters.

Given a recursively defined type like "T is a list of objects of type T," the recursive counter algorithm works without modification. A finite tree of data terminates on lists of length 0, which tells the stack of counter indexes to pop immediately. Types including records, such as trees with data at each node, and union types, such as different types for inner nodes and leaf nodes, are more complicated but essentially the same mechanism.

This method has a slight disadvantage for compression, since list lengths of very different cardinality are folded into a single stream of numbers, and it is not usable for random access data. Our reason for excluding it is that iterating over nested loops,

```
for p1 in event.particles:
    for p2 in event.particles:
        do_something(p1, p2)
```

requires the stack of counters to be copied so that we can unwind to the previous state at the end of an inner loop. If sublists like p2 can be saved and used outside of the loop, even more bookkeeping is required. This adds a great deal of complexity and significant degradation in performance.

Finally, the missing expressiveness of not having recursively defined types can be made up for (and more) by adding a pointer type. Recursively defined types only permit arbitrarily deep trees, not circular references; pointers do both.

The fifth and last method for describing hierarchical data in columns is the one used by Dremel and its open-source implementation, Parquet. Rather than introducing counters, offsets, or event identifiers, whose values scale with the size of the dataset or the nested collections within it, Dremel/Parquet introduces "repetition levels," which only scale to the depth of the nested structure. This is especially valuable in a storage format, since it encodes all list structure with a minimum number of bits. A similar concept, "definition levels," sparsely encodes missing data, including empty lists. (Definition levels are *required* to allow lists to be empty.)

In principle, this encoding could represent recursively defined types by allowing the repetition level to take any nonnegative integer (in a variable-width format, for instance), but Parquet does not currently take advantage of this capability. This encoding is not good for random access, but it is intended as a storage format with variable-width packings and compression.

Many of the developers of the Parquet Java and C++ libraries are also developers of Apache Arrow, an in-memory format intended for iterative analysis. It is significant that they did not choose to represent hierarchical structure in Arrow using repetition and definition levels, but rather offset columns. This is the method we will focus on for the remainder of this document.

The PLUR typesystem

To simplify the discussion and implementation of columnar, hierarchical data, we consider only four kinds of types:

- **Primitives:** fixed-width quantities, mostly numbers of different resolutions (32 and 64 bit integers, unsigned integers, single and double precision floating point), but also potentially logical types (booleans, 1 or 8 bit) and fixed-width *characters* (ASCII, Unicode UTF32, but not UTF8).
- **Lists:** homogeneous, ordered, arbitrary-length sequences of some other type. LIst(T) is a type constructor that takes any type T and produces a list of that type. Lists can be nested. Strings are lists of bytes and optional types (also known as "nullable" or "maybe monads") are lists with a maximum length of 1.
- **Unions:** set of possible types {T}, of which an instance has only one type (tagged at runtime). Known in type theory as a "sum type," this allows lists to be slightly non-homogeneous, containing several predetermined types, such as "electrons," "muons," and "taus," rather than just "particles."
- **Records:** unordered set of named, typed fields, such as "particle containing energy (float) and mass (float)." May also be represented as an ordered tuple of unnamed, typed fields, in which case the names are just the fixed indexes of the tuple. In type theory, this is a "product type," complementary to unions in the sense that an instance contains a member of every type it contains (a particle has energy *and* mass, not energy *or* mass).

We use the acronym "PLUR" (Primitives, Lists, Unions, and Records) to refer to this typesystem and the data representation described in the next subsection. The PLUR typesystem is simpler than the typesystems of most programming languages and most data serialization systems, but we believe that this minimal set is sufficient for useful work. In fact, most HEP analysis could be done without unions (PLR), but excluding unions would force ugly workarounds in the few cases when they are needed.

Many data structures for high-performance algorithms cannot be directly constructed in PLUR because PLUR only *represents* data, it doesn't necessarily serve it in the most optimal way for fast algorithms. For example, a HashMap<K,V> can be represented as a PLUR List(Record(key:K,value:V)), but it doesn't have O(1) lookup unless an actual hashmap is constructed from the list. We assume that an implemented typesystem includes metadata to indicate when a PLUR type should be used as-is or used to construct some runtime object.

Also, note that our record types do not have names. The fields are named, which is a minimum requirement for duck typing and structural typing, and this typesystem is suitable for languages that dispatch functions that way. Nominal typing, such as a system that would distinguish a “Point3d” with attributes “x”, “y”, “z” from a “Vector3d” with the same attributes (e.g. to apply an affine coordinate transformation to the first and a linear transformation to the second) can be satisfied by supplying distinct names as type metadata or names generated from content to emulate structural typing in a nominally typed language. This may be used, for instance, to distinguish a UTF8String from a List(byte).

A particularly important “runtime type” is Pointer<C>. Pointers are represented by PLUR integers with type metadata identifying another column— the pointer “references” that data[2]. These references might be outside of the data structure, such as an event collection with links to external calibration data, or it might be within the data structure, like particles pointing to the jets they are associated with, and it might even be circular, creating graph structures. The availability of pointer types makes it unimportant for the typesystem to include recursively defined types, since the tree structures this feature would allow can be made with pointers instead.

Just as runtime types are an abstraction layer above PLUR, concerning themselves with which functions accept a given set of arguments or how to build data structures for high-performance algorithms, PLUR is concerned only with representing hierarchical data as columnar arrays, not the representation of the arrays themselves. It is an abstraction layer over the arrays, which may be in memory, on disk, in a simple file like Numpy’s “.npy” format, in a complex file like HDF5 or ROOT, or generated on demand. They may be compressed, big-endian, or in a variable-width format, as long as a name and an index returns a primitive value of the correct type.

Thus, PLUR is not a file format or even an in-memory representation like Apache Arrow. To the degree that PLUR represents list structure, sparse union structure, and record structure the same way that Arrow does, Arrow data may be used with tools that interpret PLUR data, but it is a one-way compatibility.

[2] This is less powerful than pointers in C, which can point to any type of data. However, that flexibility is rarely desired and may be the data structure equivalent of limiting code to WHILE and not GOTO.

The PLUR data representation

An object that can be described in terms of Primitives, Lists, Unions, and Records can be represented as a set of arrays with unique names. These generated names are usually appended to a user-provided prefix, so that they can fill different corners of a namespace containing many PLUR objects. If this prefix ends with a “/” and the namespace is a filesystem, then each columnar array becomes a file. We use the word “column” to refer to the array description— name and partition— and the word “array” to refer to the numerical data they contain.

A sometimes-useful feature of the generated column names is that they losslessly encode the data type. Thus, an object can be reconstructed from its column names and data even if the type information is lost or not saved in a separate metadata system.

Reusing a set of arrays in a different data structure amounts to viewing them with a different set of column names. One level of name-indirection can save petabytes of disk space by sharing common components among different versions of a dataset, slims, or soft skims.

The following algorithm packs a data structure *D* of type *T* into named arrays. Recursion begins with *D* = the whole object, *T* = the whole type, and *N* = the user-provided prefix.

- **If *T* is a primitive**, append *D* to an array named *N*, creating it if it doesn’t exist yet.
- **If *T* is a list** with contained type *T’* and length ℓ, find an array named *N* + “-Lo” (list offset). If it does not yet exist, create it with a single element 0. Then, select the last element *e* from this array and append $\ell + e$ to the end of the array.
  Next, iterate through each item in the list and apply the rule for *T* = *T’* with name *N* = *N* + “-Ld” (list data) on the element as *D*.
- **If *T* is a union** with possible types $T_1, \ldots, T_n$ and *D* has actual type T_t, find or create an array named *N* + “-Ut” (union tag) and append *t*.
  Next, follow the rule for type $T = T_t$ with name *N* = *N* + “-Ud” + *t* (union data *t*) on the same data *D*.

- **If *T* is a record** with field names and field types (N_1, T_1), …, (N_n, T_n), follow the rule for each pair N_f, T_f using name $N = N$ + "-R_" + N_f (record field N_f), type $T = T_f$, and the field data as *D*.

Primitive types have a one-to-one relationship between the data to be represented and the array contents, though boundaries such as event boundaries are ignored. Lists and unions require integer-valued arrays to represent their structure, though the maximum value in the union tag array is known and can be used to pack the data, usually within one byte (or less!). Records have no structure other than contributing to column names. For this reason, records are the most malleable types— we can add or remove layers of record structure (e.g. turning raw 4-momentum components into 4-momentum objects) without touching the data.

The transformation of a hierarchical object into columnar arrays and its type into column names is reversible: see arXiv:1708.08319 for the algorithm.

The unions described here are Arrow's "sparse unions" in that they don't pad arrays for data that do not exist. Since the columns associated with each type branch can have a different offset, random access to a union requires an offset array "-Uo" to jump to the right element of the type branch. This offset array can be generated from the tag array "-Ut" (see the same paper for an algorithm).

Lists in PLUR may have several different representations. The list offset "-Lo" described above provides compatibility with Arrow, but a list size "-Ls" would provide compatibility with ROOT; the two are easily interconvertible. In addition, we make use of list begin "-Lb" and list end "-Le" arrays which can be derived from a list offset:

```
begin = offset[:-1]     # Pythonic slicing
end = offset[1:]        # (in Numpy, these are views, not copies)
```

giving us additional flexibility to skip list elements or step through them out of order. This will be relevant for soft skims and database-style indexing.

Datasets and partitions

Given a set of named arrays, a “dataset” is a mapping from names to names, where the new names are consistent with a PLUR type structure, and all offsets exist[3]. Datasets are usually derived from other datasets so that they have semantic meaning as well.

A dataset need not be a list of objects, such as a list of events. It could be a record (possibly containing lists) or even a single primitive. But typically, and for all data derived from ROOT, the topmost type constructor is “List.”

Lists may be split among independent processing units to perform calculations in parallel; each partial list (split at an integer number of items in all substructure) is called a partition. The partition number must be part of the column lookup, and the dataset description must include all partition numbers.

Naturally, calculations that are scattered among partitions can’t include loop-carried dependencies— the product of these calculations is usually a monoid, such as an aggregation, a histogram, or new dataset columns with the same partitioning as the parent. Monoids can be incremented in-place within a partition and combined across partitions.

Repartitioning small partitions into larger ones by integer factors is easy— concatenate every two, every three, every ten, etc. Any other repartitioning is hard— more so than with flat tables. In general, one needs to find the item boundaries, which are further complicated if the list is described by begin/end arrays, rather than an offset or size array.

Dataset manipulations

Once a PLUR object has been produced or converted from another format, it has a *natural* dataset, the set of column names and data type used in the production or conversion. However, this is only a starting point.

[3] That is, if a list offset “-Lo”, list begin “-Lb”, list end “-Le” minus one, or union offset “-Uo” contain a value *e* that would be applied to another array, that *e* must be less than the length of the target array.

Most calculations that take a dataset as input will implicitly "project" it, which is to produce a new dataset by dropping unused record fields (also known as "slimming"). Most languages have a dot-syntax for extracting record attributes, and the name of the attribute must be known during parsing. With a simple type-check, all record fields that might be used in a calculation can be determined before executing the code, and hence a much smaller projected dataset can be derived from the source by just ignoring some columns. This vastly reduces the data to be loaded before processing, in a way that is similar to ROOT's "SetBranchStatus."

Projections may also be explicit, for instance to bind to a new C++ class with fewer fields or fields with different names than the one used to generate the data, as in schema evolution. If a new version of the class has additional fields or fields with different data types, then a new columnar array will need to be generated according to some rule, such as generating a constant, estimating the quantity from other values, or casting one type to another. These operations do not affect the unaltered fields and thus schema evolution is a minimally invasive operation, which may be done on the fly.

Updating a dataset changes its content by adding fields or changing values in fields, such as applying a new calibration or fixing an incorrectly computed value. Usually, one wants to keep the original dataset for continuity and A/B tests. To update a columnar dataset, one only needs to supply the new columns, new dataset name/version identifier, and new mapping of names to names. Since the new version of the dataset and the old version share most of their data, the old version may be kept in perpetuity. The only limitation is that lists cannot increase their lengths, only drop items or reassign items from one list to another at the same level of structure.

A semantically more complex list operation partitions a once-continuous list into shorter sublists within a data structure (not to be confused with the "dataset partitions" described in the previous section). As an example, one may fit a dataset of events into a larger dataset of runs, luminosity blocks, and events (a list of lists of lists). The events and their contents are unchanged, but the continuous list is broken into luminosity block chunks, which are further broken into run chunks. This has semantic consequences for code executed on the old and new datasets:

```
    for event in events:
        do_something(event)
```

becomes

```
    for run in runs:
        for luminosityBlock in run.luminosityBlocks:
            for event in luminosityBlock.events:
                do_something(event)
```

all without touching the event data. This technique will be relevant for implementing zone maps in the database-style indexing section below.

Data can be skimmed (dropping events) in two ways: soft skimming leaves the original unfiltered data as it is and adds two columns, an alternate list begin ("-Lb") and list end ("-Le"); hard skimming actually copies the original. Alternate begin/end arrays act to update the dataset like any other column, but operate on list structure. The "event list" is the begin array; the end array is used to quantify sublist length when events are not abutting (an offset array is not sufficient). This operation can be applied to any list, including particle lists within events.

A hard skim is a traditional HEP skim: a literal copy. There are a few circumstances when one might want to do this heavyweight operation (in terms of processing time and storage of the result), such as downloading only the skimmed events and repartitioning skimmed data so that it can be accessed more quickly. Columnar structure does not impede traditional skimming, but it doesn't help, either. During a hard skim, one must also choose columns to select (a hard skim is also a hard slim), whereas a soft skim takes all columns implicitly.

Zero-deserialization scanning

One particularly attractive feature of representing hierarchical data in arrays is that traditional data structures such as C++ classes and std::vectors do not need to be filled. Generally, the way programmers and data analysts interact with hierarchical data is through objects, but this is a syntactic convenience. The data themselves do not need to be contiguous objects in memory at runtime.

If data analysts were to thoroughly understand the splitting procedure, they could write analysis code that accesses the array elements by index. However, this is too much to ask—it's not their job to dig into low-level details of data representation. They need to focus on the physics and statistical issues, for which a high-level object-oriented interface is most convenient.

Instead of converting data from arrays into object structures, we can convert user code so that it accesses array elements by index. This code conversion could be seen as a kind of inverse of data deserialization, changing the code to meet the data as it is, rather than changing the data to meet the assumptions of the code. This technique and its results are discussed at length in arXiv:1708.08319. In short, it can produce very fast analysis functions because restructuring data into objects is a much slower operation than the simple predicates that are applied to most events.

Database-style indexing

Another feature that columnar data opens up is the possibility of database-style indexing. While zero-deserialization scanning accelerates access by some constant multiple, database-style indexing can change the time complexity of the search, e.g. from linear to logarithmic. Depending on the query, the gains could be orders of magnitude.

Indexing techniques, however, are usually designed with tabular data in mind. Hierarchical data adds the complication that there are complex relationships among arrays of different lengths. A major simplification, however, is that our data are immutable, so the cost of generating an index is less relevant and there is no cost associated with maintaining the index— only (minor) storage considerations.

The key problem is that physicists want to select *events* based on the properties of *particles* contained within those events. Using particle data to form a predicate on events implies some sort of aggregation: does the physicist want the maximum value among particles in each event, the sum, or something more complex? It's often a maximum subject to some filter on particles, such as "maximum p_T track with $\chi^2/N_{dof} < 10$," which is a different quantity than “maximum p_T track with $\chi^2/N_{dof} < 5$.” It would not be possible to generate an index on all such possibilities.

However, the majority of the selection power is in the cut on raw particle properties: “maximum p_T track.” Given that an event contains such a particle, a fine-grained predicate can be later applied to verify that it is exactly what the physicist wants. Thus, we can provide coarse indexes that do the bulk of the data reduction, which then have to be followed up by fine-grained predicates applied to the reduced data. We need to use inequality theorems such as “the maximum p_T track is greater than or equal to the maximum p_T track with $\chi^2/N_{dof} < 10$.” The physicist user will need to supply the coarse selection and the fine-grained predicate to ensure that the intersection of these is desired, since coarse selections can’t, in general, be derived from procedural code[4].

Probable candidates for indexes include first, second, third, and fourth largest p_T (momentum transverse to the beamline) for every particle type in the event, the MET (missing transverse energy), and possibly number of primary vertexes and maximum displaced vertex. Note that p_T maxima imply existence of particles up to a count of four, and usually a physicist only trusts that a particle was properly reconstructed if its momentum is greater than a given threshold. More complex quantities could be indexed, such as masses of particular combinations of particles, particles in reconstructed decay chains, or particles with “blessed” isolation cuts, but these would be analysis group-specific, generated at the request of the interested parties. All of these quantities are functions of an event, not a particle, so the indexes are applied to the table of *events*.

Now that our quantities to index are functions of the events, we can adapt standard indexing techniques to them. Below, we will describe zonemaps, bitmaps, and two kinds of sorting, though other techniques may also apply.

Soft sorting with permutation indexes

Sorting a dataset provides an absolute index: if a user is interested in events with “highest muon $p_T > 50$” and the events are sorted by maximum muon p_T, we can safely find the first

[4] This is one area where a declarative analysis language would help. Since declarative code is algebraic, coarse selections could be algebraically derived from fine selections in the code. With procedural code, one could identify specific patterns, such as an “if” statement whose predicate only includes equalities and inequalities (no functions) appearing directly within a “for” loop.

event with a $p_T > 50$ muon by a bisection search and iterate from there to the end. Only matching events would be passed on to the analysis function.

The situation is complicated by the fact that different analysis groups are interested in different quantities: physically sorting the list of events by maximum muon p_T improves searches for events with high muon p_T but worsens searches for events with "second-highest jet $p_T > 100$." However, the list does not have to be *physically* sorted. Leaving the data as-is, we can create several "soft sorted" versions of the same list by computing the permutation that would sort it using the `argsort` function.

Walking through a PLUR-represented list involves offset-lookups; walking through it in a sorted order with a permutation array adds only one more layer of indirection. Instead of finding the `i`th event and its particles by

`eventdata[i]` and `particledata[particleoffset[i]]`,

we would find the `i`th *sorted* event and its particles by

`eventdata[permutation[i]]` and
`particledata[particleoffset[permutation[i]]]`.

There is another hidden cost in accessing data this way: disk and/or memory accesses are non-sequential. The relative impact of page-misses versus skipping unnecessary events depends on the specifics of the problem and remain to be seen.

It is also fairly common for analyses to be interested in two or more particle collections, such as "highest muon $p_T > 50$ AND second-highest jet $p_T > 100$." It would not be possible (without a degradation in performance) to resolve the intersection while iterating over the results: the intersection (or union for logical OR) must be performed before iteration. Intersection and union algorithms with hashmaps are linear in the size of the sets; Numpy's `intersect1d` and `union1d` are *O(n log n)*, but they return a sorted list of indexes, which might be preferred because the pass over PLUR arrays would then be sequential.

It must be emphasized that these operations— sorting, intersection, and union— would be performed on individual column *partitions*, not the entire dataset, which could span many independent processors. Sorting 8 MB of random numbers takes hundreds of milliseconds at most. Furthermore, our datasets are immutable, so the sorted indexes do not need to be updated, once built.

Zonemap indexing

Zonemaps are an approximate index which summarize data in a zone (interval) of events, where the zone size (number of events) is a configurable parameter. Zonemaps typically store the maximum and minimum values of the indexed quantity, so that a search for "events with $p_T > 50$" can skip zones whose maximum p_T is less than or equal to 50. Some quantities, such as p_T, might only need one-sided zonemaps, since the low-p_T end of the particle spectrum is dominated by noise[5]. Some quantities that are selected in very narrow ranges, such as J/ψ mass, would be entirely unhelped by zonemaps.

Zonemaps can also be hierarchical. Whereas the first layer of zones allow a scan to skip unwanted events, another layer can be built on top that skips unwanted first-layer zones. With enough layers, the time to find a rare event can be logarithmic in the total number of events, with the base of the logarithm being the depth of the tree of nested zonemaps.

The advantage that a zonemap provides over a sequential scan, guarded by a predicate, is that the predicate is only evaluated for a minority of the events. That is, a zonemap with zone size 100 can replace 100 floating point p_T comparisons (1 per event) or more (1 per particle with multiple particles per event) with just 1 or 2 (just maximum or maximum and minimum). If zones are too large or the desired selection is not sufficiently rare, the zone map provides no advantage, but only adds 1 or 2 comparisons per zone.

Sometimes, evaluating the per-event predicate is not the bottleneck— loading the data is. Fetching a columnar array from disk or over a network can be considerably more expensive than the floating point comparisons, due to limitations in physical hardware. If the data are being fetched from disk, the ideal scenario would be to make zone sizes equal to disk pages (4 kB) and memory-map the predicate arrays, so that array indexes are only touched if a selection passes the zone map. If the data are being fetched over a network, the zonemap has no natural granularity, but selections should be batched to minimize network requests for the zones that match.

[5] On the other hand, analyses desiring the *absence* of a particle ("exclusive searches") might find value in a minimum p_T.

Unfortunately, it may be impossible to match zone sizes to disk pages for hierarchical data. Zones are intervals of *events*, but the predicates and subsequent analysis operates on *particles*, which in general have different offsets. A disk page of event-level quantities may be more or less than a disk page of particle attributes, often dramatically different. Moreover, we can't make zones apply exactly to disk pages of particle attributes by varying the number of events per zone because they must correspond to an integer number of events. However, if we make the first layer of zonemaps significantly smaller than the size of disk-pages per particle, only a minority of them would straddle disk pages, resulting in fewer but not zero unnecessary disk-page loads.

Varying the number of events in a zone to improve zone size for a particular type of particle could worsen the zone size for another type of particle. Events contain rare particle types (such as electrons and muons) and abundant particle types (such as jets), and tuning zone size for both might not be possible. It may be that rare and abundant particles are served by selections at different layers in the zonemap hierarchy. All of these are issues that are only raised because our data are not tabular, and are therefore interesting from a theoretical point of view.

Once a zonemap has selected a subset of events for further investigation, a sequential scan must be executed on that subset. The zonemap selection must be efficiently delivered to the scanning procedure or the advantage would be lost. One way to do that would be to integrate the zonemaps directly into the PLUR code transformation (described in the previous section). Zonemaps can be added to a dataset as just a few new columns that partition the data similar to the run/luminosity block/event example above.

That is, we add the following three new columns:

```
zone-Lo
zone-Ld-R_ptmax
zone-Ld-R_events-Lo
```

and drop one:

```
events-Lo
```

to make a dataset in which the `List(Event)` becomes a `List(Record(ptmax=float64, events=List(Event)))`. In addition, code is transformed from

```
for event in events:
    do_something(event)
```

to

```
for zone in zones:
    if zone.ptmax > 50:
        for event in zone.events:
            do_something(event)
```

All of the event content and structures within the event are preserved and none of the original columns need to be modified, and yet the data structure is changed: events are no longer a continuous stream, but short sublists within the list of zones. This procedure can be applied to arbitrarily many levels of zonemaps.

The method we just described integrates zonemap-selection with the user's analysis code, and thus they will be executed at the same time. In some circumstances, it may be preferable to select all zones before executing any user code, such as when the purpose of the zonemap is to minimize network traffic: we want to batch all array-subset requests before submitting them.

In this case, we

1. use the zonemap to find a set of event indexes to select,
2. download the dataset's list begin ("-Lb") and list end ("-Le") arrays for the particles of interest (or make them out of list offsets ("-Lo") or list sizes ("-Ls") if that is how these particle lists are characterized),
3. look up the list begin and end values for only the events of interest and make a new list begin and end for the subset,
4. use these ranges in a batch request from the data source, over the network,
5. use these begin/end arrays in place of the originals. The user code does *not* need to be modified.

With either method, the malleability of data in a columnar format is key to performing these operations quickly.

Bitmap indexing

Another technique for accelerating selections is to reduce event-level quantities to a few bits per event and perform bitwise operations to identify matches. Unlike zonemaps, bitmaps characterize every event, and are therefore more useful for less-rare searches. They are also useful for searches that target mid-range values, such as J/ψ and Z masses or central η regions, rather than extremes. Bitmaps are a particularly natural choice for selecting events by the detector trigger that caused them to be saved and reconstructed, since these quantities are already booleans.

Bitmaps first reduce a quantity to a small number of bits. Triggers (early decisions in detector read-out, boolean-valued) can be represented by one bit each, and a user's predicate like "muonTrigger OR jetTrigger" would be translated into "triggerBits & (muonTriggerMask | jetTriggerMask)", where the parenthesized quantity is a constant that can be precomputed.

A quantity for which inequalities are interesting can be binned, such as 16 bins for η (pseudorapidity) from $\eta = -0.8$ to $\eta = +0.8$.The first and last bin can serve as underflow and overflow bins, taking η values beyond the window. Each value of η fills only one bit, producing patterns like 0000001000000000. A user's predicate like "$-0.4 \leq \eta < 0.4$" gets translated into the following mask: 0000111111110000. Predicates that don't exactly line up with bin edges must be expanded to be slightly too inclusive, and a predicate within the user code must remove the edge cases.

Some quantities are only interesting in very narrow ranges, for which the above would be too coarse. J/ψ and ψ' masses, for instance, have well-known values that are nearly equal. Custom windows that separate them define categorical values, for which categorical equality, rather than ordinal inequality, is interesting: an analysis function either selects J/ψ particles or ψ' particles, not both in the same pass. In this case, distinct values can be mapped to distinct bit patterns, like 11 of the 16 possible integers expressible with four bits (representing, e.g., η, ρ/ω, ϕ, J/ψ, ψ', ψ'', $\Upsilon(1S)$, $\Upsilon(2S)$, $\Upsilon(3S)$, $\Upsilon(4S)$, and Z dimuon mass windows). These bit patterns would be matched by integer equality, rather than bit masks.

It's important to note that bitmasks don't change the *number* of comparisons that must be made, only the speed with which they are performed. A bit mask is much faster to compute than a floating point comparison, and might require less data to be loaded. Furthermore, bit masks can be vectorized, whereas the nested structure of user analysis code often prevents vectorization. So while bitmasks provide an acceleration technique, they don't change the time complexity of searches, the way sorting and zonemaps do.

Hard sorting particles for better disk utilization

All of the methods described above select a subset of events for further study. Preselecting with an index reduces the number of predicates to be evaluated in the analysis function, but what if the bottleneck is disk-loading, rather than predicate-evaluation? A selection that eliminates more than a factor of 1024 avoids some unnecessary disk page-reads, assuming 4 kB disk pages and 4-byte data values. However, index-selections that eliminate far less than that, such as an index-selection that only removes half of the events, would hit every or nearly every page because the gaps between selected events are not nearly large enough to include whole pages.

Under these conditions, all effort spent indexing the data is wasted: skipping events doesn't save time. However, we can avoid this situation by concentrating useful events on the same pages.

To do this, we need to *physically* ("hard") sort the events. Primarily, the particle columns must be sorted because most of the data is found in particle attributes. For the PLUR data representation to be valid, we must maintain the contiguous order of particles within each event, but move all particles associated with an event as a group such that events have the preferred ordering. This involves some tricky indexing, but it is possible.

Next, we need to decide on an order of events. Sorting by muon p_T improves searches for events with a constraint on that quantity, but worsens searches for events with a constraint on another quantity, such as jet p_T. Fortunately, the columnar representation allows us to physically sort muon-related columns by muon p_T and jet-related columns by jet p_T, so we do not need to pick a favorite attribute for the whole event. We do have to pick a favorite attribute for each particle type, but in HEP, the choice is clear: it should be the p_T of that particle.

Finally, to use the physically sorted columns, we need list begin ("-Lb") and end ("-Le") arrays that translate the original order to the new order. With these in place, the new, sorted dataset appears to the user exactly like the original one, but when they step through events in the original order, array values are extracted in the sorted order. Access with this dataset will always be non-sequential.

The advantage is that preselection techniques such as the soft sorting with permutation indexes, zonemap-skipping, or bitmap filter yield results that are concentrated at the end of the physically sorted arrays, assuming the user selected for high p_T. Though out of order, they are likely to touch and therefore load fewer disk pages, even for preselections that eliminate only half of the events. In a regime dominated by disk access bottlenecks, this could be the difference between database-style indexing having no impact to having a significant impact.

Concluding remarks

This document describes some of the many possibilities that are made possible by representing hierarchical data as columns, and further managing the data with columnar granularity. These techniques promise to dramatically reduce data duplication while optimizing access. Zero-deserialization iteration and database-style indexing become possibilities when the data are physically represented by simple, numerical arrays.

In this document, we also described a specific mechanism for expressing complex types as arrays: any types describable as a combination of Primitives, Lists, Unions, and Records (PLUR) can be split into flat arrays. Knowing the rules for how these arrays are to be interpreted, they can be manipulated to emulate dataset updates, slimming, skimming, and other restructuring without any data duplication. Data analysts no longer need to decide upfront which attributes they might need in their analyses, nor do they have to repeat a months-long filtering job when they guess incorrectly. Sysadmins no longer need to provision storage for multiple versions of the same data with different cuts or different versions of the same data.

This style of access works best on a shared server, since it maximizes the degree to which different views of the data can reuse the same physical copy. There are some reasons why copying may still be necessary— to get local access or repartition after a highly selective cut— and these “hard skims” would have the same difficulties that data management has today, no worse.

However, given the benefit of shared data stores, we advocate for the development of query-based analysis, in which all analysis functions, including final plots, are produced by querying the shared data store. High throughput and low latencies would be critical for such a system to compete with local hard skims, so we seek to take advantage of zero-deserialization scanning and database-style indexing as much as possible.

These techniques are currently under study as research, but with the goal in mind of developing a usable product. We are in touch with the physicist user base and computer science researchers to make this project a reality.