Fast Access to Columnar, Hierarchical Data via Code Transformation

Jim Pivarski, Peter Elmer
Physics Department
Princeton University
Princeton, NJ, 08544
pivarski@princeton.edu, peter.elmer@cern.ch

Brian Bockelman, Zhe Zhang
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588
bbockelm@cse.unl.edu, zhan0915@huskers.unl.edu

Abstract—Big Data query systems represent data in a columnar format for fast, selective access, and in some cases (e.g. Apache Drill), perform calculations directly on the columnar data without row materialization, avoiding runtime costs.

However, many analysis procedures cannot be easily or efficiently expressed as SQL. In High Energy Physics, the majority of data processing requires nested loops with complex dependencies. When faced with tasks like these, the conventional approach is to convert the columnar data back into an object form, usually with a performance price.

This paper describes a new technique to transform procedural code so that it operates on columnar data natively, without row materialization. It can be viewed as a compiler pass on the typed abstract syntax tree, rewriting references to objects as columnar array lookups.

We will also present performance comparisons between transformed code and conventional object-oriented code in a High Energy Physics context.

Index Terms—Big data applications; Automatic programming; Data analysis; Scientific computing; High energy physics instrumentation computing

I. MOTIVATION

Analysis procedures can be roughly categorized by whether they are easily expressed as transformations of data tables into new tables, by SQL or an SQL variant, rather than a general-purpose programming language. We must emphasize that “easily expressed” is subjective, and therefore illustrate with progressively more complex examples.

Suppose we have High Energy Physics (HEP) data in a table with the following schema.

```
CREATE TYPE PARTICLE FROM
  STRUCT<pt: DOUBLE,
    eta: DOUBLE,
    phi: DOUBLE>;

CREATE TABLE events (  
  eventId INT,
  muons ARRAY<PARTICLE>,
  jets ARRAY<PARTICLE>,
  UNIQUE KEY eventId
);
```

Often the first exploratory action in a HEP analysis is to plot a distribution of the highest particle pt per event. We could do that in SQL by exploding the muons into a virtual table and aggregating the maximum pt, grouped by eventId. If the execution engine loses track of the fact that unique values of eventId are contiguous in the exploded table, the query would unnecessarily search the entire dataset for eventId matches. A Python or C++ program would not.

The second action a HEP analyst usually takes is to compute masses from pt, eta, and phi of pairs of particles to see if any were the decay products of a short-lived intermediary. Candidates for such a decay would always be found in the same event, so this query in SQL would involve a full outer join on two exploded muons tables or exploded muons and exploded jets, subject to equal eventId. In this case, it is even more important that the uniqueness of eventId is tracked, since a join across a distributed database can be very expensive.

HEP data manipulations only get more complex from there. The problem is that the hierarchically nested schema is a poor fit to the relational model for which SQL was designed. Tree-like, nested data are much easier to deal with in a procedural or functional language, particularly one with object-oriented features.

However, sequentially iterating over a large set of objects and applying a user defined function to each can be much slower than a database query, even when parallelized. Databases enjoy several advantages: (1) indexing by primary keys, (2) columnar data layout, and (3) code manipulation (“term rewriting”). Indexing is beyond the scope of this paper, but columnar data layouts are increasingly being used to store hierarchical data. Code manipulation of general purpose programming languages is only common at the compiler level, but data analysis frameworks are beginning to use Just-In-Time (JIT) compilation and metaprogramming in their workflows.

ROOT[1], the HEP analysis framework, was first to store hierarchical data structures as columns. The ROOT file format (in “split mode”) stores primitive attributes adjacent to one another on disk, e.g. all muon pt values, followed by all muon eta values, etc. Google independently developed Dremel[2], on which Apache Parquet is based. Spark 2.0 and Apache Arrow extend this concept to data in memory, and each has

1\sqrt{2p_1\text{pt}p_2\text{pt}^F(\cosh(p_1\eta - p_2\eta) - \cos(p_1\phi - p_2\phi))} for p_1 and p_2
seen substantial increases in data throughput when switching to a columnar layout, especially for processes that access only a few of the attributes.

JIT compilation is central to machine learning tools like H2O[3] and Theano[4], Julia[5], a scientific programming language, as well as ROOT. To our knowledge, however, none of these manipulate the user’s code to match the structure of columnar data.

In this paper, we propose a technique to transform user code to more quickly access columnar data in memory. This technique is inspired by similar methods used on SQL, but SQL tightly controls all loops over data, making the transformation easier. We find that it can be performed on general user code if an Abstract Syntax Tree (AST) with type information is provided, or in conjunction with a type-inference pass over an untyped AST. The user code does not need to walk through the data in order, though it would benefit from fewer cache misses if it is sequential with a regular stride, just like any low-level procedural program.

In fact, code transformation could be viewed as a general alternative to object deserialization: when faced with user code that expects objects but the data are in another form, one could either transform the data to fit the code’s expectations (deserialization) or transform the code to fit the data. Deserialization is the more common choice, and the only practical one for complex data formats, but code transformation can be much faster. When manipulating code, the data format is only interpreted once at compile-time. When manipulating data, the format is interpreted once per object, in addition to extra memory allocations and copying.

II. DATA REPRESENTATION

We begin by describing the scope of data types we are considering. To simplify the transformations, we restrict this set as much as possible while still being useful. In particular, the data types described here only encode data; they don’t determine how data are used, such as which functions can be legally called on them. However, an interpretation layer can be overlaid on this representation without affecting the format or code transformations.

A. Type System

The set of possible types is generated by four constructors:

- **Primitive**: fixed byte-width booleans, integers, floating point numbers, and possibly characters.
- **List**: arbitrary-length list of another type, including nested lists.
- **Union**: represents objects that can be one of several types (“sum types” in type theory).
- **Record**: a container mapping field names to types (“product types” in type theory).

We call this type system PLUR, an acronym of the four constructors. A complete type schema in PLUR is a tree of Lists, Unions, and Records whose leaves are Primitive types.

One thing to notice is that this system does not allow for recursively defined types. For instance, one cannot make a “Tree” Record containing Trees. All data structures have a finite maximum depth. The pointer mechanism described below makes up for this deficiency.

Another thing to notice is that Records are not named, as structs and classes are in C++. This allows for flexible structural typing, in which the only criteria for a type to be accepted as the argument to a function is that it has the minimum set of fields. If stronger type safety is desired, extra metadata can be added to the type description to indicate type names and/or subclass relationships.

As a concrete example, names can distinguish a raw List[byte] from a String or even a UTF8String, which should be interpreted in different ways by user code. Similarly, nullable data can be represented as an Option[X], which is a List[X] that only allows zero (empty, null) or one (filled) elements.

A particularly useful type is Pointer[X], which is an integer interpreted as a List index. In addition to having a type name that distinguishes it from ordinary integers, this type would have metadata identifying another List in the structure; dereferencing the pointer returns the item at the List index. With this mechanism, not only can tree data be constructed, but graphs with cycles as well. It also provides a way to compactly represent selected events from large datasets, since a physicist’s “event list” is a List(Pointer(Events)).

B. Objects to Arrays

Any data described by the PLUR type system can be converted into a set of named arrays. The array names all begin with a user-provided prefix, so that the user can manage arrays representing different objects in a single object store, zip file, HDF5 file, or Python dictionary. Names are generated by a recursive walk over the type structure, and in fact the type schema is losslessly preserved in the names themselves. This may be useful if an external metadata system is not available to store the type schema.

Data are packed into arrays by recursively applying the following rules to a given name \( N \) and type \( T \) in the structure, where \( N = \) the prefix and \( T = \) the whole type schema at the root of the data structure.

If \( T \) is a **Primitive**, append the primitive value to an array named \( N \), creating it if it doesn’t yet exist.

If \( T \) is a **List** with contained type \( T' \) and length \( \ell \), find an array named \( N + \text{-Lo} \) (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 \( T' \) with name \( N + \text{-Ld} \) (list data).

If \( T \) is a **Union** with possible types \( T_1,\ldots,T_n \) and the datum has actual type \( T_t \) (where \( t \in [1,n] \)), find or create an array named \( N + \text{-Ut} \) (union tag) and append \( t \).

Next, follow the rule for name \( N + \text{-Ud} + t \) (union data \( t \)) and type \( T_t \).

If \( T \) is a **Record** with field names and field types \( (N_1,T_1),\ldots,(N_n,T_n) \), follow the rule for each pair \( N_f, T_f \).
(where \( f \in [1, n] \)), using \( N + \text{“-R_”} + N_f \) (record field \( N_f \)) as a name and \( T_f \) as a type.

A Record does not generate any arrays to represent its structure (as Lists and Unions do); the connection between fields in an array is entirely encoded in the naming convention. To ensure that array names can be properly parsed, field names must not contain the character “_” (or a different delimiter should be chosen).

Be sure to include empty/trivial arrays for types that were not touched due to missing data (Lists that are all empty at a given level or Union type possibilities that never occur in the data) or make the reading procedure insensitive to missing arrays. A simple way to include all arrays is to create them with a first pass over the type schema and only append to them in the pass over data.

C. Arrays to Objects

The procedure described above losslessly stores the type schema in array names and the data in the arrays. To demonstrate this, we describe a procedure for reconstituting objects and types from the arrays and their names.

First, select arrays whose names begin with the prefix and pop the prefix from their names. For each of these arrays \( a \), create an index \( i_a \) whose initial value is 0. Then recursively apply the following rules.

If only one array exists and its name is the empty string, then the type is a Primitive. Take the value at index \( i_a \) and increment \( i_a \) by 1.

If one array name begins with “-Lo” and all others begin with “-Ld”, then the type is a List. Take \( a[i_a + 1] - a[i_a] \) as the length \( \ell \) of the List and increment \( i_a \) by 1 for the array \( a \) that begins with “-Lo”.

Pop the “-Ld” from the beginning of all other array names and apply the rule for that set of arrays \( \ell \) times.

If one array name begins with “-Ud”, and all others begin with “-Ut”, then the type is a Union. Take \( a[i_a] \) as the tag \( t \) for the datum and increment \( i_a \) by 1 for the array \( a \) that begins with “-Ut”.

Pop the “-Ud” + \( t \) from the beginning of all other array names associated with \( t \) and apply the rule for that subset.

If all array names begin with “-R ”, then the type is a Record. Partition the set of arrays by field name \( N_f \), pop the “-R ” + \( N_f \) from the beginning of the array names, and apply the rule separately for each partition to fill each field.

If any other configuration of arrays and names is encountered, the arrays are malformed. If indices go beyond the lengths of the arrays or do not perfectly end on the last element of each array, then the arrays are malformed.

D. Random Access and Redundancy

The reason that List structures are represented by data offsets (arrays whose names end in “-Lo”), rather than List lengths, is to permit random access. For instance, if we had a List(List(float)) named with prefix “x” and we wanted the float at index \((i, j)\), we would compute

\[
x-Lo-Ld[x-Lo-Lo[x-Lo[0] + i] + j]
\]

At each level, the contents of the “-Lo” array are the starting indices of the next-deeper structure.

In fact, if we want to reconstruct only one object in a List, we simply apply the arrays-to-object algorithm for that List’s “-Ld” arrays with all indices starting at \( i_a = i \).

Random access is a useful feature, but this offset array is not optimal for compression or variable-length integer encoding algorithms. If we wish to store or transmit a List structure in the fewest possible bytes, it should be converted into a corresponding size array (suffix “-Ls”). For all \( i \in [0, \text{length of } x-Lo - 1] \),

\[
x-Lo-Ls[i] := x-Lo[i + 1] - x-Lo[i]
\]

and vice-versa. (The offset array is one element longer than the size array because we include the initial offset 0, which simplifies the code transformation in the next section.) A nice option would be to transfer or store List structures as size arrays for less redundancy, then expand as offset arrays for random access.

The Union structure, as described so far, is not random accessible. Data arrays for each type possibility \( T_t \) are only filled if a datum of that type is observed, so each type possibility must be indexed by a different offset. The arrays-to-objects algorithm described in the previous subsection avoids this issue by walking over the data sequentially.

To make Unions random accessible, we need to add a union offset array “-Uo”. This can be generated from the tag array “-Ut” by

\[
i_t := 0 \text{ for all } t \in [1, n]
\]

\[
\text{for } i := 0 \text{ until length of } x-Ut \text{ do}
\]

\[
t := x-Ut[i]
\]

\[
x-Uo[i] := i_t
\]

\[
i_t := i_t + 1
\]

\[
\text{end for}
\]

Now we can access Union objects randomly: to reconstruct a Union at index \( i \), we find the tag \( t \) at \( x-Ut[i] \) and follow the arrays-to-objects algorithm on the set of arrays named with the corresponding “-Ut”, all starting at index \( i_t \) given by \( x-Uo[i] \).

This union offset array is entirely redundant: it can always be generated from a tag array. Therefore, it never needs to be transmitted or stored.

E. Relationship to Apache Arrow

Readers familiar with Apache Arrow might recognize this object-array encoding[6]: it is identical for equivalent Arrow types. Arrow additionally defines null-value masks for each type and headers for seeking to the memory address of each array, rather than assuming a name-based lookup.

With the appropriate translation between PLUR types and Arrow types, array names and Arrow memory addresses, the code transformation procedure described in the next section can be used unmodified on Arrow data, and therefore Pandas[7], R[8], or SparkSQL[9] DataFrames that are internally represented as Arrow. Alternatively, one could expand
our code transformation procedure to cover the Arrow specification natively.

III. Code Transformation

An important feature of this data representation is that a structure of any type can be expressed by a single integer, the index that would be used to start the arrays-to-objects algorithm. A type and an array name prefix uniquely specifies a set of arrays, so the only new information at runtime are locations within those arrays.

Only one index is required because:
- a Primitive value is located at one index in its array;
- a List’s length can be computed from one index \((a[i_a] + 1) - a[i_a]\) and all of its contents are derived from the value of its offset array at that index;
- a Union is specified by a tag from its tag array and an offset from its offset array, which occur at the same index. All of the contents are derived from the offset;
- a Record is just a bundle of fields with no structure array. However, the contents of all fields in a Record object start at the same index.

Therefore, any function that operates on Primitive, List, Union, and Record objects, no matter how complicated, can be replaced with a function that operates on integer indices. We can shape the code to match the data format, rather than shaping the data to match the object-oriented vocabulary of the code.

To see this as an alternative to deserialization, consider a scenario in which all arrays are in a binary file on disk that has been memory-mapped to look like an array. When the transformed code accesses a structured object, it is actually reading values from disk. No additional representation of the data is created; the new function knows what to do from the raw byte values and proceeds as the old function would have with objects. We have skipped deserialization.

The technique described below can be applied to code written in any language, but all of the explicit examples are written in Python code because our implementation[10] transforms Python code.

A. General Strategy

Unlike SQL, procedural and functional programming languages can assign and possibly reassign variables, extract substructure as new variables, loop over substructure, and pass objects to other functions, where they are identified with new names. It is not sufficient to only transform symbols that coincide with prefix names, since parts of the data structure may be spread among user-defined variables. To avoid missing code that needs to be transformed, we must track the PLUR type of all symbols in the function.

We therefore need a typed AST, in which PLUR types and array name prefixes are associated with any expression nodes that hold non-Primitive PLUR type. (Knowledge of other types is not necessary but not harmful.) The prefixes must be propagated with the type because different parts of a data structure may have the same type but different values, and the correct array references need to be injected into the code. Code transformation and type propagation may be performed in a single sweep.

There are several constraints on the code to be transformed. It cannot create or change PLUR objects, which is reasonable for an analysis procedure. It can use external functions, but if a non-Primitive PLUR object is passed as an argument to such a function, the code for that function must be accessible so that it can be transformed, too. If the same function is called in different places with different PLUR types/names as arguments, multiple versions of the transformed function will be generated. These external functions cannot be created or passed dynamically; their code must be accessible for conversion during the transformation sweep.

The following AST nodes must be transformed:
- symbol reference, which might be a PLUR object;
- assignment, which might pass PLUR type and prefix to a new symbol;
- list subscript (square brackets in most languages), which can dereference a PLUR List;
- attribute subscript (dot in most languages), which can dereference a PLUR Record;
- function calls, which can pass PLUR types and prefixes to a new function, widening the scope of the transformation sweep to include that function, and may return a PLUR type/prefix that must be tracked;
- special functions, like len (List length) and isinstance (check type) in Python, which have to be handled in special ways when called on PLUR types.

Any use of a PLUR type that isn’t specially handled must be treated as an error to avoid misinterpretations. Since this transformation sweep is performed on an AST, often before the transformed code is further transformed by a low-level compiler, these errors would appear to the user as compilation errors, preferably with line numbers and meaningful error messages.

B. Required Transformations

Symbol references are leaves of the AST and therefore first to be transformed in the recursive walk. In the original function, an identifier that refers to a PLUR object becomes an identifier for its integer index, so a symbol reference becomes array extraction.

For Primitive and List types:

\[ x \text{ (referring to object)} \rightarrow a[r][x] \]  

where array is the only array associated with a Primitive and the list offset associated with a List.

Unions immediately resolve to their specific type (unless you require the user to “unpack” opaque Union objects with a function). This is most easily handled by (1) creating a new symbol in the transformed code to hold the tag value and (2) checking each tag possibility, returning the offset of the appropriate one:
\[ x \rightarrow t := \text{tag}[x]; \text{if} \ (t == 0) \ \text{off}[x] \ldots \]

This is more difficult in languages like Python, which do not allow assignment in an expression (the symbol reference can appear anywhere in an expression); see our Python implementation \cite{10} for the work-around.

References to Record symbols do not require any transformation at all, though they are reinterpreted as indices to be passed to fields.

**Assignment** merely passes the PLUR type and associated array names to a new symbol. In the type-inference pass, this means adding the type information to a symbol table.

Assignment is more complex if you try to handle pattern matching, such as Python’s `tuple unpacking` (see “Flourishes” below).

**List subscripts** replace a List index with a Primitive value if it is a List of Primitives or the index of the next substructure down if it is any other type. This can be performed in two steps: (1) transform
\[ x[i] \rightarrow \text{off}[x + i] \]
and then (2) transform the result of this as though it were a symbol reference (rule described above) using the List’s item type.

An attempt to subscript any PLUR type other than a List is an error. The above only works if the index resolves to integer type, not (for example) a Python `slice`. Handling slices would be considered a flourish.

**Attribute subscripts** extract a field from a Record by name. In most languages, the field names are syntactically required to be a string known at compile-time with constraints on the characters (usually, “-” is not allowed).

We do this transformation in two steps, like the list subscript above: (1) transform
\[ x\.\text{fieldname} \rightarrow x \]
and then (2) transform the result of this as though it were a symbol reference (rule described above) using the selected field type.

An attempt to subscript any PLUR type other than a Record is an error.

**Function calls** can either include non-Primitive PLUR types in their arguments or not. If a function call does not reference any non-Primitive PLUR types, it can be left as-is (after transforming the arguments).

If it references non-PLUR types, then we must obtain the AST for that function and propagate PLUR types through it, starting with the argument types. If the function has previously been transformed, then the transformed copy may be reused only if the argument types (and array prefixes) are the same as before. We are treating all functions as though they were type-parameterized in every argument because different array prefixes require different code transformations.

If a function is recursive, then this process will only terminate if previous transformations of the function are saved and reused for the same type arguments. It is not merely an optimization to reuse transformed functions!

A function may return PLUR type; the transformed version is either a Primitive or an integer index that must be propagated from the point of the function call in the original code.

**Special functions** that return meaningful data about non-Primitive PLUR types must be handled on a case-by-case basis. As with any function call, if a non-Primitive PLUR type is an argument, the function’s identity must be known during the transformation sweep. Below are two important cases.

**len**: (get length of List) must return the length when given a PLUR List, which is computed as
\[ \text{len}(x) \rightarrow \text{off}[x + 1] - \text{off}[x] \]
(applied without transforming the argument \(x\), or by reverse-engineering an applied transformation).

**isinstance**: (check type) must return true if the argument has a given type, false otherwise. Since references to non-Primitive PLUR types have all been replaced with integer indices, a naive application of the `isinstance` function would say that they are all integers, contrary to a user’s expectation.

The type or types to check must be known as literal names, not computed references, during the transformation sweep. If the argument to check has List or Record type, the whole expression should be replaced with true or false depending on whether the desired types are in the set of types to check. If it is a Union, it must be replaced with a tag check.

**C. Flourishes**

Depending on your needs, you can provide more or less support for PLUR objects in the transformed code. The only invariant is that the transformed code must either work exactly as object-oriented code would or fail to be transformed (compilation error).

1) **List Overflows**: One special case requires special attention: what to do about List index overflows? In the minimal transformation described above, PLUR List overflows behave like C array overflows, in that they return undefined results. The errors are less obvious and therefore more dangerous than typical C array overflows because PLUR values just beyond a List’s boundary belong to the same attribute in the next List, so they probably have the right scale and distribution, subtly biasing analysis results.

A simple way to eliminate mistakes like this is to add a range check to the transformed code. Indices that fail the range check should raise a runtime exception. It would be correct, but many of these range checks would be unnecessary and would slow down processing. For instance, the list subscript could be in the body of an `if` statement where the user did range-checking manually, or it could be in a `for` loop bounded by the list length (a very common case), or it could be a `foreach` node (Python `for`) that has been transformed from objects to indices (another flourish).

Since we are performing type-inference anyway, we could track intervals of validity for expressions that appear in list subscripts. This is known as a dependent type check, and for our case the validity endpoints should be symbolic: literal integers or \(T\) symbols for the length of List type/array prefix.
The symbolic algebra includes only inequality operators, addition and subtraction, logical operators, and restrictions in if-else blocks. If the expression in a list subscript is not provably between 0 (inclusive) and $\ell_T$ (exclusive), a compilation error tells the user about the possibility of List overflows and refuses to transform the code.

2) **Pythonic Indices**: Negative index handling in Python would be another flourish, which complicates the above.

3) **Fixed-size Arrays and Matrices**: PLUR Lists can have any length, which includes a constant length, at the expense of having redundant offset arrays. The size array for a fixed-length List is filled with identical values, which is very compressible, but must be expanded to full length for random access.

It would be possible to add the concept of a fixed-length List to the type system, along with a naming convention so that this type can be encoded in array names, rather than arrays. In place of an array lookup in the transformed code, one would insert a literal multiplication.

4) **Eliminating Zero-Lookups**: As an optimization, one can statically identify array lookups that always return zero: the first element of every list offset array is zero, and the outermost list offset array is always evaluated at its first element. Without explicitly removing it, this unnecessary code would be executed at every step in an iteration over a List. (PLUR-unaware compilers cannot remove it.)

5) **Type Constructors**: As stated above, passing a non-Primitive PLUR object to any unrecognized function is a compilation error. This especially includes constructors for dynamic objects like Python lists and dictionaries, since we cannot statically track where they are used. However, you may wish to allow some immutable objects to include PLUR objects, such as Python tuples to use Python tuple-unpacking in assignments. You then become obliged, however, to track these objects as containing PLUR types.

6) **Pattern Matching**: More generally than Python’s tuple-unpacking, your target language may feature pattern-matching, and you may need to track PLUR types through syntactical structures like these.

7) **Equality and Order**: PLUR objects have a natural definition for value-based equality, so you may want to support the $\equiv$ operator as a recursive object comparison. Reference-based equality (“is” in Python) is even easier: just verify that the indices are equal (in Python, replace “is” with “==”).

For identical types, there may be a natural ordering as well: Python defines an ordering on structures like Lists and Records. You can similarly support inequality operators and the “sorted” function.

8) **Fallback to Object Materialization**: One way to support particularly difficult cases, such as external functions without code transformation, is by materializing objects using the arrays-to-objects algorithm. This gives up on zero-copy efficiency, but it may be worthwhile to mix fast code with expressive code. Then, rather than new features allowing specific cases of user code to run at all, new features would allow the user code to run faster.

9) **Runtime Types**: As described in Section II-A basic types described only in terms of Primitive, List, Union, and Record generators should be augmented by higher-level interpretations of these types, distinctions that are relevant for runtime behavior, not data packing. These runtime types would be implemented at this level.

For instance, a String or UTF8String type would be admissible to functions that capitalize and lower, but List(byte) would not. The former would be implemented as a special function, like len and isinstance. Pointers should either be transparently dereferenced (can cause infinite loops) or opaque with a special function or syntax.

IV. **Implementation**

We are developing PLUR and code transformation tools for use in a HEP query system. The code is in an early but usable state as documented by the text on the GitHub READMEs.

Both are Python projects, and the code transformation implemented in the PLUR package is written in Python, transforming Python code. The short-term goal is to build a query system that uses Python as its query language, with a long term goal of developing a declarative query language on top of PLUR.

A. **Pythonic PLUR**

All of the “required transformations” (described above) have been implemented except for function calls and the isinstance special function. Of the “flourishes,” zero-lookups have been eliminated and tuple-unpacking assignments has been implemented. The problem of implementing Union symbol references has been solved by passing nested closures through the transformation sweep.

This PLUR package optionally passes the transformed code to Numba for an even greater speedup. Although Numba’s conversion of Python code to native bytecode results in a much larger speedup factor than PLUR’s transformation step alone, the Numba step wouldn’t be possible without the PLUR step. The PLUR transformation replaces opaque Python objects with integers and array lookups, which fit squarely in Numba’s vocabulary.

B. **Columnar Data from ROOT**

The ROOT file format is the universal standard for HEP. Fortunately for our application, ROOT can store hierarchical data in a columnar fashion, smoothing the transition to PLUR. Most of these arrays can be imported without modification. The one exception is that Lists are represented in ROOT with size arrays, rather than offset arrays, so we apply the transformation described in Section II-D.

ROOT is a C++ framework providing an object-oriented interface, so we also had to bypass part of its deserialization procedure to get the raw arrays directly into PLUR. In a fork of the ROOT project, we added methods to TBranch (ROOT’s data column) to extract arrays after decompression but before object materialization, as well as methods to expose
these as Numpy arrays in Python. These enhancements, known as BulkIO, are being merged into the main ROOT repository with the intention of being included in ROOT version 6.12.

## C. Performance Studies

The goal of this and the HEPQuery project is to make HEP analysis easier to develop and faster to run. We believe that broad type abstractions like PLUR and concise syntax like Python’s are easier for data analysts to work with, especially when they are exploring a dataset and focusing on physics and statistics issues, rather than programming. We also intend to make the system as fast as or faster than idiomatic, object-oriented C++, so that analysts don’t have to translate analysis code from a high-level language to a low-level one just to process a large dataset in a reasonable time. This is the allure of databases: SQL is as abstract as set theory and yet it is also convenient for performance testing now.

For this reason, we have been performance-testing PLUR all through its development, to avoid introducing performance gaffes at an early stage, and now perform an even-handed performance comparison between transformed PLUR code and industry-standard C++ in ROOT. We also break down the differences as much as possible to show where they come from.

We consider four sample analysis functions, two of which (“max pt” and “mass of pairs”) are from the motivation of this paper. Listings are given in Figure 1.

The first, “max pt,” is an example of a query that would be difficult but not impossible in SQL. Instead of exploding a muons table and grouping by a unique identifier, we select a muon by maximizing pt and then plot its eta. This is even more awkward in SQL, but very common in HEP.

In the listing, note that best, which is a muon object, is initialized as –1 in Python and nullptr in C++. The current implementation of the code transformer is missing a concept of “undefined variable.” It is also missing a thorough check on pre-transformation data types, so it doesn’t complain about a variable being –1 in some cases and a Record object in others. Post-transformation, it is an array index, so –1 is allowed and doesn’t correspond to any real objects. This is a leaky abstraction and will be corrected in future versions, but convenient for performance testing now.

The third function, “mass of pairs,” would require a full outer join in SQL but is a nested for loop in Python and C++, carefully indexed to avoid duplication. This kind of “unique pairs” loop is very common (often with a selection, e.g. requiring opposite-sign charges), and the mass formula is one of the most frequently executed in HEP.

The fourth, “py sum of pairs,” is a diagnostic of the third. As we will show below, the mass calculation is the slowest of the sample functions, and it was unclear whether the nested indexing was responsible or the complex formula. This function has the same nesting structure but a simpler formula (one that is occasionally useful in HEP).

```
max pt in Python
maximum = 0.0
for muon in event.muons:
    if muon.pt > maximum:
        maximum = muon.pt
fill_histogram(maximum)
```

```
max pt in C++
floata maximum = 0.0;
for (i=0; i < muons.size(); i++)
    if (muons[i]->pt > maximum)
        maximum = muons[i]->pt;
fill_histogram(maximum);
```

```
eta of best by pt in Python
maximum = 0.0
best = -1
for muon in event.muons:
    if muon.pt > maximum:
        maximum = muon.pt
        best = muon
        fill_histogram(maximum)
```

```
eta of best by pt in C++
floata maximum = 0.0;
Muon* best = nullptr;
for (i=0; i < muons.size(); i++)
    if (muons[i]->pt > maximum)
        maximum = muons[i]->pt;
        best = muons[i];
        fill_histogram(best.eta);
```

```
mass of pairs in Python
for i in range(n):
    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) -
            cosh(m1.phi - m2.phi))
        fill_histogram(mass)
```

```
mass of pairs in C++
for (i=0; i < n; i++)
    for (j=i+1; j < n; j++)
        Muon* m1 = muons[i];
        Muon* m2 = muons[j];
        double mass = sqrt(2*m1->pt*m2->pt*;
            cosh(m1->eta - m2->eta) -
            cosh(m1->phi - m2->phi));
        fill_histogram(mass);
```

```
pr sum of pairs in Python
for i in range(n):
    for j in range(i+1, n):
        m1 = event.muons[i]
        m2 = event.muons[j]
        s = m1.pt + m2.pt
        fill_histogram(s)
```

```
pr sum of pairs in C++
for (i=0; i < n; i++)
    for (j=i+1; j < n; j++)
        Muon* m1 = muons[i];
        Muon* m2 = muons[j];
        double s = m1->pt + m2->pt;
        fill_histogram(s);
```

Fig. 1. Sample analysis functions in Python (before code transformation) and object-oriented C++, showing only the body of the loop over events.

We ran each analysis function on a 5.4 million event dataset of simulated Drell-Yan collisions (a realistic physics sample, one of about a dozen that might be involved in a real HEP analysis). The tests were performed on an i2.xlarge instance on Amazon Web Services, which features a large, fast SSD (hundreds of MB/second, not a bottleneck in our tests).

Both tests, Python and C++, used ROOT to read the same data files. However, the Python test used the BulkIO feature to extract columnar array data without constructing event objects. The C++ code used the conventional interface, which further deserializes the columnar arrays as C++ objects.

One thing that ROOT’s conventional interface does not do (but in general a framework might) is allocate memory for the event and muon objects during the event loop. Memory allocation recognized as a performance killer early in ROOT’s development, so ROOT’s default is to preallocate a buffer of 1000 muons and overwrite them at the start of each event, reallocating the buffer if necessary. (This never happened in our sample; the maximum number of muons was 36, which is far above the mean of 1–2.)

Also, to avoid decompression or physical access being a dominant contributor to all tests, we prepared an uncompressed sample and loaded it into Linux page cache with vmtouch[14]. This lets us see the differences due to other factors more clearly.

Both tests were single-threaded and had plenty of working
The results of the study are shown in Figure 2. The shorthand “ROOT” signifies a conventional ROOT workflow, using C++ user code, and “PLUR” is our workflow using the BulkIO interface in ROOT to expose columnar data as arrays and then run transformed Python code on those arrays.

The four colors signify variants on these workflows. “ROOT full dataset” means we let ROOT fill all 42 attributes of the Muon objects, which is clearly unnecessary for our functions. The event processing rate for this case is 0.4 MHz, regardless of the content of the function. Reading and filling the objects overwhelms all other factors.

“ROOT selective on full” uses ROOT’s built-in mechanism to avoid filling all attributes in the objects, but still used the 42-field object definitions and original data file. The event processing rate is 1.29 MHz, regardless of the function. Handling all of the attributes still dominates.

“ROOT slim dataset” performs the same selective read on a specially prepared dataset and object definition that has only 3 fields: pt, eta, and phi. We now see different rates for the four functions: 6.68, 5.96, 3.56, and 6.34 MHz.

“PLUR full dataset” is the only test in this batch that uses transformed Python code. It accesses the full dataset; the slim dataset yields similar results because this method is unconcerned with unused columns. The four functions are processed at 17.9, 12.1, 6.09, and 17.2 MHz, respectively, considerably faster than the C++.

The slowest of the four functions is “mass of pairs.” Unlike the first two functions, this involves a doubly nested loop over muons to find distinct pairs, as well as a much more complex formula. The fourth function has the same loop structure without the complex formula, and it is as fast as the single loops. Upon further investigation, we found that the trigonometric and hyperbolic cosines account for the majority of the time spent in this function.

The factor that the ROOT tests and PLUR test had in common is that they both extracted data from ROOT files and used part of the ROOT framework to read them. PLUR can operate on any arrays, regardless of whether they came from ROOT, so we performed another test in which we extracted all the data as Numpy arrays and read them from SSD disk or memory into the transformed Python code. These results are presented in Figure 3 as “raw SSD disk” and “raw memory.” The “PLUR full dataset” measurements are redrawn in this plot to clarify the change in scale.

When the data are in memory, we see only the calculation cost with no I/O. The effect of the slow mass calculation is dramatic: “mass of pairs” runs at 12.8 MHz while “pT sum of pairs” runs at 56.2 MHz. Reading from this SSD disk and performing the calculation is 1.6–3.3 times slower than just performing the calculation.

This is relevant because the HEPQuery system is being designed with a two-layer cache: frequently accessed columns will stay in memory, less frequently accessed columns will bepaged to an SSD disk like this one, and infrequent or first-time requests will be extracted from ROOT files using BulkIO.

V. CONCLUSIONS

We have demonstrated that it is possible to transform code to meet a data format, rather than deserializing data to fit the code’s expectations, as long as JIT compilation is available. We have also demonstrated that, once transformed and compiled, a Python analysis function outperforms the same function written in idiomatic C++.
On the surface, this is almost unthinkable—C++ is supposed to be much faster than an abstract language like Python—that’s why HEP data analysts go to great effort to port their Pythonic explorations into C++. However, our code transformation and Numba’s bytecode compilation have eliminated Python’s disadvantages and given it an advantage not shared by the idiomatic C++, skipping object deserialization. This procedural code is one step closer to behaving like a database query.

In HEP, we plan to use this tool as part of a database-like query server, delivering analysis results in response to queries written in Python. The PLUR data representation gives us the ability to represent large Lists of HEP events, possibly with deeply nested hierarchies and pointer cross-references, as well as random access to non-event data. Possible uses for non-event data include lookup tables, calibration constants, boosted decision trees and neural nets.

The use of columns for data granularity, rather than files that contain columns, opens up possibilities for better efficiency in data storage. It is fairly common for a HEP dataset to undergo several revisions, in which derived quantities are progressively corrected with new knowledge. Currently, each new version requires as much disk space as the previous one, which is wasteful if only a few attributes change. With column granularity, a new dataset can be defined using some new and some old columns, like structural sharing in immutable data structures or version control systems like git. Additionally, analysts would be more free to experiment with different event selections if each is just a single column that points to a shared set of publicly accessible events, rather than a copy of a subset of rows.

But perhaps most importantly, the usefulness of this technique is not limited to HEP. The need for complex loop dependencies can hardly be HEP-specific. Although many industries and fields of academia are currently using SQL or languages similar to SQL for data analysis, how much is being left unexplored because the tools are not suited for the task?

It is our hope that this technique finds application in many different fields, just as the techniques of database-style analysis have inspired our work in HEP.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under Grants No. 1450377 and 1450323. The authors wish to thank Philippe Canal for his expert help in understanding the ROOT file format and ROOT I/O subsystem.

REFERENCES

[1] R. Brun and F. Rademakers, “ROOT: An object oriented data analysis framework,” Nucl. Instrum. Meth., vol. A389, pp. 81–86, 1997.

[2] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” in Proc. of the 36th Int’l Conf on Very Large Data Bases, 2010, pp. 330–339. [Online]. Available: http://www.vldb2010.org/accept.htm

[3] The H2O.ai Team. (2015) “H2O: Scalable Machine Learning. Version 3.1.0.99999.” [Online]. Available: http://www.h2o.ai

[4] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” arXiv e-prints, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, “Julia: A fast dynamic language for technical computing.” CoRR, vol. abs/1209.5145, 2012. [Online]. Available: http://arxiv.org/abs/1209.5145

[6] The Apache Arrow team. (2016) “From the Apache Arrow wiki: Physical memory layout.” [Online]. Available: https://github.com/apache/arrow/blob/master/format/Layout.md

[7] W. McKinney, “Data Structures for Statistical Computing in Python,” in Proceedings of the 9th Python in Science Conference, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[8] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, “Spark SQL: Relational Data Processing in Spark,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’15. New York, NY, USA: ACM, 2015, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2723797

[10] Jim Pivarski. (2017) “PLUR version 0.5.” [Online]. Available: https://github.com/diana-hep/plur/releases/tag/0.5.0

[11] ——. (2017) “HEPQuery version 0.1.” [Online]. Available: https://github.com/diana-hep/hepquery/releases/tag/0.1.0

[12] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, ser. LLVM ’15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833162

[13] Brian Bockelman, Zhe Zhang, and Jim Pivarski. (2017) “ROOT-BulkIO and Numpy interface”. [Online]. Available: https://github.com/jpivarski/root/tree/root-bulkio-fastread-v2

[14] Doug Hoyte. (2012) “vmtouch — the Virtual Memory Toucher”. [Online]. Available: https://hoytech.com/vmtouch/