Developing a Declarative Analysis Language: LINQToROOT

Gordon Watts[1,*]
[1]Department of Physics, University of Washington, Seattle, Washington, 98195, USA

Abstract. The HEP community is preparing for the LHC's Run 3 and 4. One of the big challenges for physics analysis will be developing tools to efficiently express an analysis and able to efficiently process the x10 more data expected. Recently, interest has focused on declarative analysis languages: a way of specifying a physicists' intent and leaving everything else to the underlying system. The underlying system takes care of finding the data - powering the event processing loop – and even exactly how to most efficiently apply a desired jet selection. If this works, this would allow an analyser to test their algorithm on a small amount of data on their GPU-less laptop and then run it on a large amount of data on a server with multiple large GPU's without having to alter their code. The LINQToROOT project, started almost seven years ago, fits this model. It has been used and tested in three ATLAS published analyses. LINQToROOT is based on the Language Integrated Query system built into the cross-platform C# language. It enables writing strongly-typed queries on a ROOT's TTree's data and transcribes the data to a C++ algorithm that can run in ROOT. Recent work on this system has had two goals: improving analysis efficiency and better understanding the requirements of a declarative analysis language. For example, a good analysis language should be able to abstract away the backend – recent work has increased the possible back ends from formerly the single Windows ROOT backend to one that runs on Linux, the Windows Linux-subsystem, and an experimental one that allows for PROOF like parallel processing – all done with almost no change to the analysis code itself. Any analysis language must also be rich enough to support an experiment's data model. To test this, some experiments with the full ATLAS xAOD data model have been performed. All of this has been done while attempting to keep the project close to its original goals: quick turnaround for real ATLAS physics analysis. This work will be discussed in some detail along with thoughts and lessons that have helped shape our thinking about an Analysis Language and perhaps our approach to future physics analysis employing declarative analysis.

1 What is an Analysis Language?

An analysis language enables the researcher to concisely and clearly specify the desired data manipulations to produce plots, tables, and other results. In the context of high energy particle physics, an analysis language allows the analyser to quickly specify plots, relationships, and other data reduction steps necessary to turn reconstructed data into a physics result.

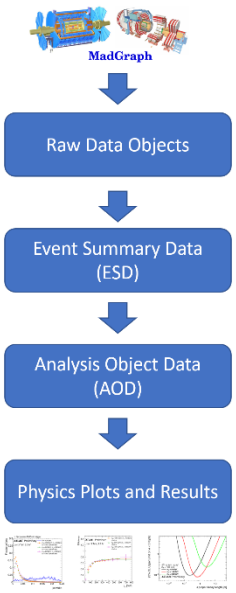

Figure 1. Raw Data or simulated Monte Carlo is processed through several phases before results appear in a paper.

The path from Raw Data to plots and tables and results suitable for publication in a modern, large HEP experiment, is one of many steps. Figure 1 shows an idealized workflow. Typically, all the steps before the last one are managed by the experiment. The larger experiments have created production systems and workflow management systems that are

* Corresponding author: gwatts@uw.edu

suited to making sure the data and Monte Carlo is processed correctly [1]. The production systems typically also carefully track the provenance of the data. All the data from each stage is typically stored in a centrally managed distributed system.

Groups performing analysis are typically small teams consisting of less than 10 people. Though, some high-profile analyses can reach 100! Each team must extract the Analysis Object Data (AOD) data from the data storage system, store it locally, code all the infrastructure to run over it, produce plots, determine systematics, tables, assemble a paper, and, finally, work through a month's long internal review process which may require re-running the full analysis chain multiple times. To say there were a large variety of approaches to this last set of steps would be an understatement, and many independent frameworks have been created by the different analysis teams (for a good introduction see [2]).

An Analysis Language attempts to address one part of this morass: from the AOD to the plots and tables. An analysis language is designed so the physicist can express the final plot or set of numbers they want to determine in a way that is well match to the data, avoiding as much of the coding boilerplate as possible (loops, declarations, etc.). A translation engine or implementation engine then converts that intent into efficient code to process the data.

A Declarative Analysis Language is an analysis language that has no explicit loops: it states step by step what the analyser wants to accomplish. Rather than specifying how to accomplish the task, the physicist specifies what they want to do. A translation engine converts that into low level loops that run in an efficient manner over the data.

For example, a declarative language would allow a physicist to ask for the p_T distribution of all jets greater than 30 GeV in events where any two jets with $|\eta| < 2.0$ have an invariant mass between 80 and 90 GeV. It is possible to specify that quite succinctly in a modern programming language. The translation engine converts that succinct specification into loops over pairs of jets, and then over jets. While it is clear what is going on in the declaration, the code will be much less clear, and significantly longer.

The situation gets even worse when we look at future computing requirements in HEP. We are rapidly moving into a world of computers with most of their compute power in co-processors, like GPU's or even FPGA's – heterogeneous computing environments [3]. Writing correct and efficient code for co-processors is much harder. One cannot imagine anyone but the most dedicated physics graduate student or someone facing a very big processing problem tackling a move to co-processors. Yet, in the USA, a substantial fraction of the CPU available to users will be in that form. And coding for one type of coprocessor is unlikely to work for another.

Several attempts were made at an analysis language before the one described in this paper. The first one was graphical based but proved to be too cumbersome: and making small changes to the analysis flow was difficult, and the abstraction was very leaky. The second was composed of bash scripts and C++ and text files for driving the analysis. There were too many different files to track to make a proper run over the data. Creating a new analysis required a large template project – and understanding what was being done was non-trivial as the analysis was split over many files. A set of requirements were arrived at after these two prior attempts along with experience from running normal C++ based physics analyses in the ATLAS experiment:

1. Succinct specification of what goes into a plot, starting from data files all the way to the plot specification.
2. The code to make an exploratory plot on test datasets and run production jobs on large datasets shouldn't be different.
3. The cost of making plot 501 after having already made 500 plots should be similar to the cost of making just one plot.
4. A single programming language should be used for everything.
5. It must use ROOT and C++ natively for the processing of the data.

The first requirement was a result of having to move between too many files to understand what went into a plot. The second came from watching many students an post-docs make exploratory plots with the easy `TTree::Draw` interface, and then spend a significant amount of time making an "official" plot in their analysis framework. The exploratory plot never had all the corrections, and the "official" plot took too long to code to be useful for exploration. The third requirement also enables exploration in a large production job. If making a single new plot is as costly as making a plot from scratch, then the analysers will be encouraged to explore using their "big" production job where all the corrections have been applied, and official selections made, etc. The fourth requirement comes from previous experience: multiple programming languages require inter-operation and multiple files. Both of which make it much more difficult to understand what is going on. Finally, at the time this project was designed there was no other apparent option for high speed processing of data stored in ROOT TTree's. Thus, for any real speed, the actual work had to be written in C++. There are other options now, like Python's NumPy libraries [4].

2 The LINQToROOT Analysis Language

The LINQToROOT analysis language was first started over five years ago with the addition of a feature called Language Integrated Query (LINQ) [5] to the C# language [6]. No extensions were made to C# for this project. Rather new libraries that plugged into the C# infrastructure were used to manage data from dataset specifications to plots was built. The guiding principle in the design was ease-of-use for the physicist. A more complete description of the project can be found in a prior proceeding [7], summarize here for context.

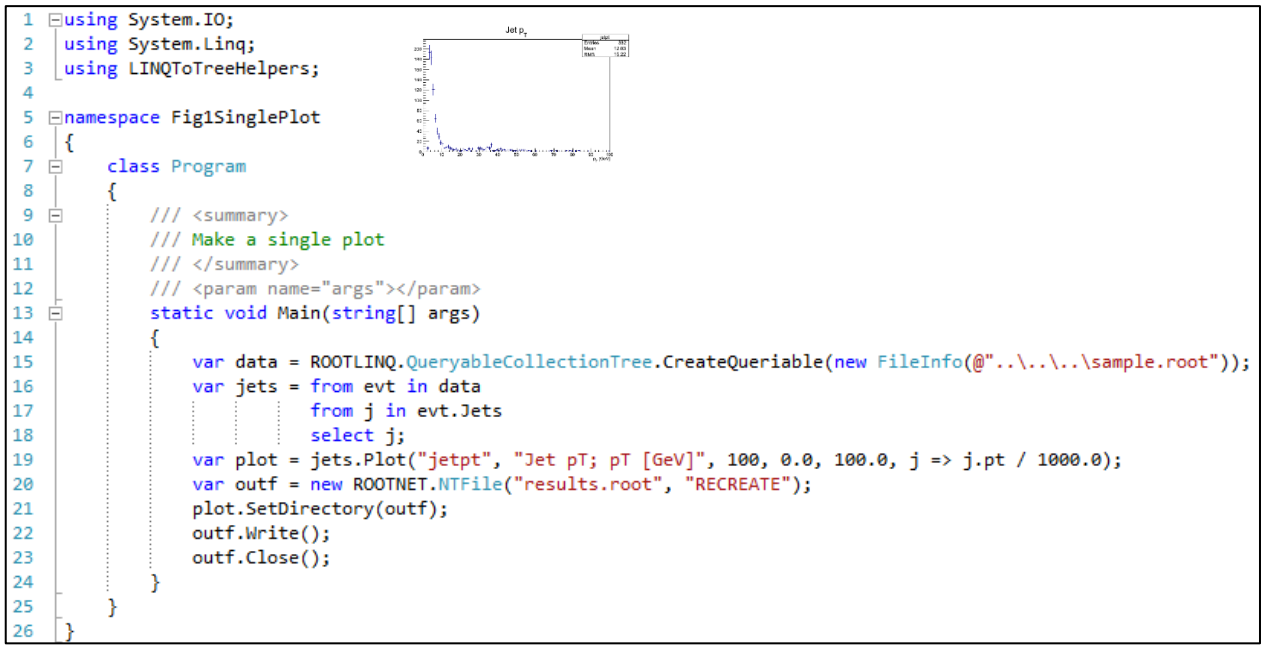

Figure 2: A simple program to demonstrate the key elements of the ROOTToLINQ analysis language. The program produces the inset plot, starting from a root file called sample.root. A .NET interface to ROOT is used to access ROOT I/O.

Figure 2 shows a short example that demonstrates many of the key concepts in the LINQToROOT analysis language. Lines 1-14 are mostly setup and standard C# boilerplate. Line 15 creates the data source from the `TTree`. Passed in is a *FileInfo* object which points to the source data file. One can pass an array of files or a dataset that refers to files located on the GRID, or even files that have been processed by a job from a GRID data source. The *data* variable should be considered as a stream of events. Lines 16-18 convert the stream of events into a stream of jets. Finally, line 19 fills a histogram with the jet's p_T. Line 19 does a lot of work behind the scenes. It writes a small C++ ROOT program to read the data and produce the histogram, compiles it, runs it, and finally gets the histogram back to the C# program. Loosely in this paper lines 16-19 are referred to as a query. These lines are also the implementation of a declarative analysis language.

A few quick comments on C#'s syntax. The expression `j=>j.pT` is a C# lambda expression. It creates a function that has an argument j and returns the value of `j.pT`. C# is strongly typed, and the function argument is of type `Jet`, which is inferred by the compiler by the type of objects being fed to the `Plot` method. For this to work, the compiler needs to know the data model for events, jets, and any other objects. This is done by scanning a ROOT file as described below.

For each entry it plots the jet p_T in units of GeV (note the automatic conversion of the string *pt* to p_T in the plot title). The variable *plot* holds a TH1F ROOT histogram object, and lines 20-23 do the usual work of storing the histogram in a ROOT output file. The ROOT.NET project provides the C# access to most of the ROOT API [8].

Before one can write the code above there is some one-time setup that must be done to configure the project. First, one must use the *Nuget* [9] tool to add the LINTToTTree-v5.34.00.win32.vc10 libraries to the project. Nuget is an open-source package manager for Windows that gives the user easy access to over 130,000 open source libraries.

Adding the library to a project in the Visual Studio IDE defines the command *Write-TTree-MetaData*. This command will scan and parse a ROOT file's `TTree`'s and generate a C# data model. Finally, some modifications can be made to the metadata generated by this command so one can use simple names like *Jets* and *pt* rather than the true names of leaves in the `TTree`. Some of this will be described in further detail later below. The author has found that one needs to modify the metadata continuously during analysis development as new branches are used – branches in the analysis `TTree`'s tend to have rather complex names. The build system automatically accounts for these changes and builds the C# data model on the fly during a build.

There is one major feature missing from this example: the ability to batch queries. The `Plot` method executes immediately. In a real analysis where 100's of plots need to be made this would be very inefficient as analysis jobs tend to be I/O bound. By calling `FuturePlot` instead, the query is queued, and a future is returned. The future can be manipulated as a monad [10], though clumsily as C# has no built-in support for monads.

It is useful to briefly discuss Line 19 further as the library does quite a bit of work in this line. Lines 16-18 build the compute graph, but line 19 completes and executes it. Figure 3 shows the shows the stages down which correspond to library components.

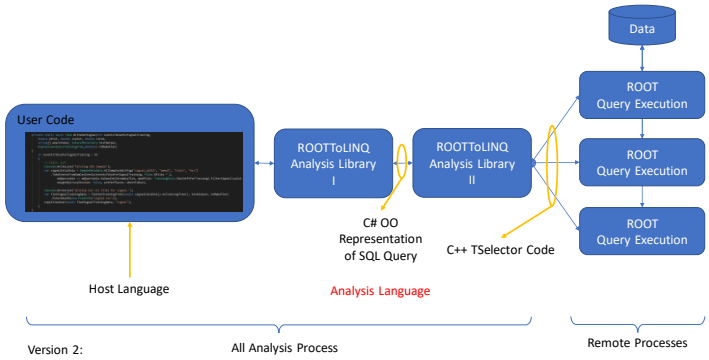

Figure 3: How a query is converted from the front end into C++ code that runs in a ROOT environment, and the resulting histograms, etc., are sent back.

Much of the heavy lifting is done by the C# language run-time, its libraries, and a few open source helper libraries designed to work with LINQ. The backend code gets an Abstract Syntax Tree (AST) that specifies every single manipulation required from the event stream to the double's that are to be histogramed. Caching is based on the AST: it is assumed that if the source files, the AST, and any data referenced by the AST are the same, then the result will be the same. If the query has been previously run, the cached previous result is returned directly. Only new queries are executed.

The second part of the back-end library then combines multiple query's AST's for any batched queries, optimizes and does things like common expression optimizations and loop invariant lifting, and then writes out a C++ `TSelector`-based program. Finally, the code is handed off to an executor that runs the query. Results are cached in a ROOT file which is handed back to the user program (as well as cached for next time). Various executors are possible. LINQToROOT includes an in-process executor, an out-of-process executor, an executor that runs multiple copies of ROOT out-of-process, and even an executor that executed across a `ssh` connection, reimplementing PROOF. At one point there existed an executor that used PROOF directly but the author was never able to make PROOF stable enough to run in this configuration.

3 Conclusions from LINQToROOT

This project was used in analyses that produce two physics papers and one conference note in the ATLAS collaboration. What follows should give a sense of scale of how it was used. Typical runs with the framework had executables processing about 2 billion events, from some 200 GRID data samples. Each analysis had input files totalling about a half a TB. The `TTree`'s being processed had about 340 leaves and were designed to be flat (no object structure). LINQToROOT can handle structured objects in the ROOT data model, but most analysers prefer to avoid this in their final analysis files. Most runs for these analyses made 400-500 plots per run and wrote them to a single ROOT output file. These jobs were also used to produce large `csv` files which were used as input for training. A TMVA bolt-on was written, but `scikit-learn` was used for the final analysis. There was one user and one developer of this tool.

Following are descriptions of the success and failures of most components of the LINQToROOT project.

3.1 The Host Language

The host language is the User Interface for the analysis language and analysis system. It is important that a powerful and capable language be chosen that allows the natural embedding of a declarative analysis language.

C# was chosen as the host language for two main reasons. First the Language Integrated Query (LINQ) feature which provided a declarative syntax and libraries for processing complex data structures. LINQ is extensible to any data source. And second because C# contains structures that make it possible to treat code as data. A small change in a declaration meant that a lambda passed as an argument to a function is converted to an AST and passed instead. This second feature allows LINQToROOT to build up a complete AST and reason

about the AST in the backend. This was crucial for combining the queries efficiently, and then optimizing them further.

LINQ is like SQL, though arguably more complete and with a better (functional) syntax. It allows one to imply loops over objects, ordering, grouping, sorting, etc. It is declarative in the sense that one does not code the loops explicitly (e.g. no for loops are written). The query expressions are written in C#. The syntax shown in Figure 2 is syntactic sugar for a functional call-tree. Most of the analysis code is written in the functional form rather than the wordy form shown there – it is concise and once the vocabulary is learned, very expressive. Being able to write code that looks like this is a crucial requirement for the host language. The host language must naturally contain the analysis language.

The ability to treat code as an AST is not complete in C#. For example, it would be nice to code a function to select jets that could be reused in a query. However, for the query system to understand this function it needs the function as an AST, not as code. C# provides no language feature that will translate a whole code block into an AST – just a single expression. This limitation can be worked around, but it makes the code a bit harder to read.

C# is a full programming language. Recent versions are including more and more functional language features like pattern matching and, of course, LINQ. This makes it very easy to code very concise and compact and understandable code. At the same time the language's roots are imperative and strongly typed, which makes it familiar to the HEP community.

The strong typing offers several advantages. For example, the compiler would complain if you attempted to access a non-existent leaf in the data. Further, editors can use the type information to aid one in writing analysis code. This is an incredibly powerful editor feature that enhances productivity (commonly called intellisense). On the other hand, this also means that the ROOT `TTree`'s must be parsed ahead of time to build a strongly typed data model. This could be quite cumbersome even if it is well hidden by the build system.

The other big weakness of the C# language is its lack of built in monad support. LINQToROOT uses futures to represent a batched query. Consider the common operation of taking the ratio of two histograms. In sequential execution one must wait until both histograms are filled before calling `TH1F::Divide`. With futures, one can setup the call to the divide method directly, against the futures – so as soon as the original histograms are filled the divide operation will execute automatically. While possible, the code – the User Interface – is rather messy and hard to understand in C#. Short of moving to a functional language like Haskell or F# I do not know away around this. Python might have a solution using its `__getattr__` object method. The C++ generative C++/metaclasses proposal also offers some hope.

There are more mundane features that C# has that are not noticed unless missing: a full programming language with complete access to ROOT. I would conclude writing a new programming language to support particle physics analysis is almost a non-starter.

3.2 The AST Parser

The first part of the backend is basically agnostic to what system processes the data and turns the query into a plot or a count or a `csv` file. Its job is to take the AST and perform translations, combine queries, and optimize. By far most of the development time spent on LINTToROOT was spent on this portion of the library. And most of the bugs appeared here.

There are a few features that proved invaluable to making the code easy to write for the physics analyser. First, is language idiom translation. For example, in C#, determining the size of an array is frequently done using the property `.Count`. Anyone used to C# will expect this to work. For some of these this was most easily done by adding a new AST node that calculated the array size and translating all references to `.Count` to that AST node.

Leaf name translation was important too. For example, in the TTree's used in this analysis, the jet p_T leaf was `CalibJet_AntiKt4_pT`. Being able to reference this as just `pT` was invaluable for keeping the code concise and easy to read. A single XML file contained all the leaf names and the names they could be referred to as in code. As new leaves were used in the analysis, modifications would be made to this XML file.

The `TTree` used in these analyses was flat. This means the `TTree` contains separate columns for the jet η, p_T, E, etc. In the code one would really like to look at them as `jet.pT` or `jet.eta` rather than `pT[jet_index]`. In short, an object view rather than a linear array view. On the other hand, ROOT is tuned to operate on columns much faster than objects. A second XML file contained a mapping from linear arrays to objects. It allowed for pointers between objects as well (for example, a list of tracks that were near a jet).

This allows the user to write code in an easily readable form and the computer to process the data in an efficient way.

It was also convenient to be able to build a struct of other objects on the fly. Imagine a new event object that contains specially selected jets and tracks. Each of these lists is assembled using a query. And then the code uses those new collections to generate plots with further queries. These struct's are place-holders for queries and the user really is chaining queries together. The code must recognize that and chain the appropriate queries together, without executing any extra sub-queries if they are not needed.

The combination of queries allows for many optimizations. For example, it is frequently the case that many plots are made with the same set of selection cuts. When batch queries are combined the fact that the selections are common should be detected so calculations do not have be performed repeatedly. The implementation of this feature cut run-times by more than a x2 in some cases. The combination code in LINQToROOT did not depend on context – queries defined in very different parts of the analyzer's code would automatically be combined based on pattern recognition. Unfortunately, if two selections in a long list were reversed, the combination would stop at the point the select cut changed.

A generic optimizer was written at the AST level as well. It looked for common expressions. For example, if one counted the number of tracks in an event and used that number in several places in the different batched queries, it would be calculated only once and cached for later use. And obvious extension to this would have been to count the number of tracks near a single jet and cache that result for each jet. However, that type of caching was not implemented. This sort of optimization depends on the fact that no calculation has any side effects. In fact, the expression conversion would fail if a reference to an external variable was detected.

A good testing framework was crucial to make sure this portion of the code functioned as expected. The bulk of the library's almost 1200 test cases covered this portion of the library's code.

Many of the operations done on the AST felt like writing the compiler optimizer (at the expression level). Unfortunately, a common library that could have been used was not found.

3.2 The Backend Converter

The backend converter converts a query into its result, translating everything to C++. It takes the simplified AST produced by the AST parser, converts it into C++ code, does some further, simple, optimizations, and then conversion to C++ code. Finally, the C++ code is shipped to the appropriate place and executed, the results gathered, and returned.

The code for this phase was surprisingly straight forward. All the possible AST forms had to be covered, of course. For example, the ternary conditional operator (the "?" in C# and C++). The goal was to implement as much of the host language expression syntax as possible to make host language programming natural. While a lot of lines of code were required, there weren't many tricks needed.

Implementing some of the LINQ operators could be, however, tricky. The most basic is a loop. In the example in Figure 2 the `from` implies a loop over all jets in the event. It is possible to add other operators that do things like find the maximum value of a sequence. To find the maximum value in an unsorted list, each items must be examined and compared against a cached largest-value-so-far variable. This code must be generated by the converter. Imagine the extension – the analyser wants to look at the jet η for the jet with the largest p_T. In this case the emitted code that caches the max value must also cache a pointer to that jet. Sorting operators require more complex temporary storage. For example, the `groupby` operator. This allows one to group a list of objects (say, jets) by some value (are they forward or central). As you might imagine, this means keeping several lists in the emitted C++ code as the jets are looped over. A great deal of work was spent initially implementing these expressions, but once implemented they remained quite stable for the life of the project. The AST optimizer saw much more churn.

The analysis language is an abstraction. And, like all abstractions, there are places that it needs to leak. A few explicit possibilities to leak were put into the code in various places. For example, direct C++ code could be injected into the query system if the code could be treated as a side-effect-less function call. There were also provisions for loading libraries and header files, and mapping functions in C# to those C++ functions. This latter feature was how things like `sin` were mapped from the C# standard math function to the C++ standard math function. This leaky abstraction was tightly tied to the fact that C++ and ROOT objects were used on the backend. It is an outstanding problem to solve how to live in a world where one's backend might be C++ and ROOT or Python and NumPy.

The back-end executor that took the C++ code and ran it on the data files was implemented through a clean interface. This allowed new backends to be implemented as the library grew in usefulness and thus the size of the datasets it needed to process.

4 Final Comments and Future Directions

The author would classify this project as a success: it has produced real physics on a sizable dataset. However, there are several obstacles preventing it from going forward. First, it is written in C#, a language that is relatively unknown in particle physics. The language also has some features which make using it difficult (like the lack of monad support). The AST parser and optimizer could stand a rewrite from scratch using lessons learned and techniques from the computer science compiler community. The caching worked well and was much faster than running on the datasets, however, calculating the caching key was slow enough that it took 4-5 minutes when looking up 800 plot results: too long. No attempt was made to write a backend that used vector instructions or a GPU; the ability to change backends is a purported advantage to the declarative programming model.

Besides the short-comings of the current system it needs to expand its horizons. For example, the backend analysis server component could be moved to a large cluster: a query server could be built. There are two languages that seem to dominate analysis in HEP – C++ and Python. Efforts should be made to develop a system that can be used in either form.

It also became clear during this project that many services around a query service are required to make analysis simple. Data preservation – tracking how a plot is made for later reference and repeatability, for example. Managing datasets and the production of flat TTree's from an experiment's AOD format are also crucial if one wants to produce a well understood histogram. This gets especially complex if the datasets might exist at CERN or your local cluster and you'd like your code to run identically in both locations. Many of these felt like small research projects after quick attempts were made to implement them well enough for the local use. It was clear that even histogramming needed some concerted architecture work. The ability to specify arbitrary template histograms, for example, or histogram axes, would greatly simplify the design of some of the analysis and reduce the number of magic numbers in the code.

C# and LINQ are a powerful combination. Going forward, however, C++ and/or Python are really the only options in HEP. A future version incorporating the lessons learned must be designed on top of these two ecosystems. The author looks forward to tackling these challenges.

References

[1] M. Borodin, "The ATLAS Production System Evolution. New Data Processing and Analysis Paradigm for the LHC Run2 and High-Luminosity," in *CHEP 2016*, San Francisco, 2016.

[2] The HEP Software Foundation, "HEP Software Foundation Community White Paper Working Group - Data Analysis and Interpretation," arXiv, 2018.

[3] S. Campana, "The ATLAS computing challenge for the HL-LHC," in *CHEP*, San Francisco, 2016.

[4] SciPy.org, "NumPy Reference," 2018. [Online]. Available: https://docs.scipy.org/doc/numpy/reference/index.html.

[5] Microsoft, "Language Integrated Query (LNQ)," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/linq/.

[6] Microsoft, "C# Reference," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/.