Design, Development and Evolution of the ROOT System

Rene Brun[1]
CERN
1211 Geveva 23 Switzerland
E-mail: `Rene.Brun@cern.ch`

Philippe Canal
FNAL
Batavia, Illinois
E-mail: `pcanal@fnal.gov`

Fons Rademakers
CERN
1211 Geveva 23 Switzerland
E-mail: `Fons.Rademakers@cern.ch`

The ROOT system started in 1995, at a time when future software directions were unclear. Many ideas were around, many prototypes were developed and many languages were candidate to replace Fortran77, but there were many committees too. Initially started as a successor of the PAW system, ROOT has considerably evolved since its initial conception. Following the long saga with Object-Oriented databases, more emphasis has been put on data modelling, data storage and data access. This paper describes the main features of ROOT as it is today and discusses the many steps, controversies and challenges that had we had to overcome to arrive at the current situation. This long process reflects the fact that systems in High Energy Physics, designed to operate for decades, require a very close collaboration with experiments.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research - ACAT 2010
Jaipur, India
February 22–27 2010

[1] Speaker

1. The ROOT System Today

ROOT [1, 2] has become the *de facto* standard in High Energy and Nuclear Physics for the experiment independent software. Figure 1 is a sketch of the various software layers in a typical HEP experiment. All these layers use in one way or another the ROOT infrastructure. Some components use ROOT only as a data storage & retrieval system. Some use it only for data analysis and the graphics features. While PAW [3] was only a single executable module *paw.exe* that could only be extended via its limited Fortran interpreter, ROOT offers about 100 shared libraries that can be directly linked by the application, or dynamically linked at run time as soon as a class of a library is referenced. This structure of shared libraries has gradually evolved during the ROOT history such that only a minimum number of libraries are required for an application. You pay only for what you use. The *root.exe* module uses less than 20 MBytes of memory at start-up and a typical application linking with the I/O and graphics requires less than 100 MBytes of memory. Libraries are organized in major logical units, see Figure 2, reflecting the way that they are typically used and also the way the project is organized into work packages for the development.

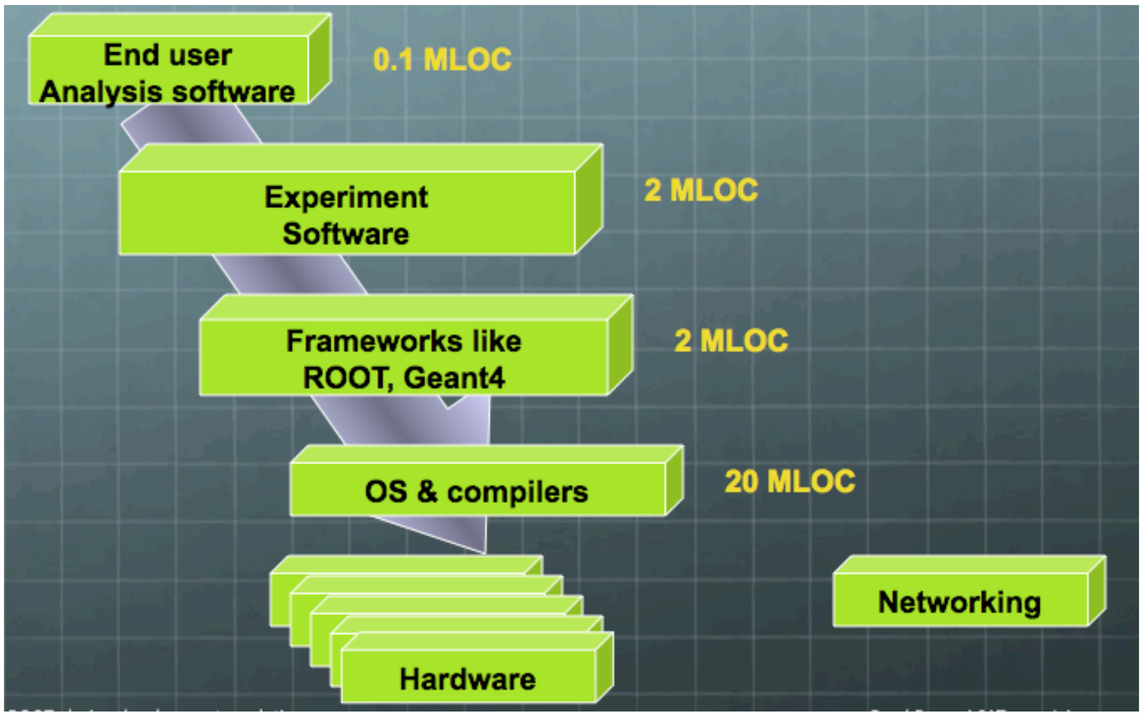

Figure 1: sketch of the various software layers in a typical HEP experiment.

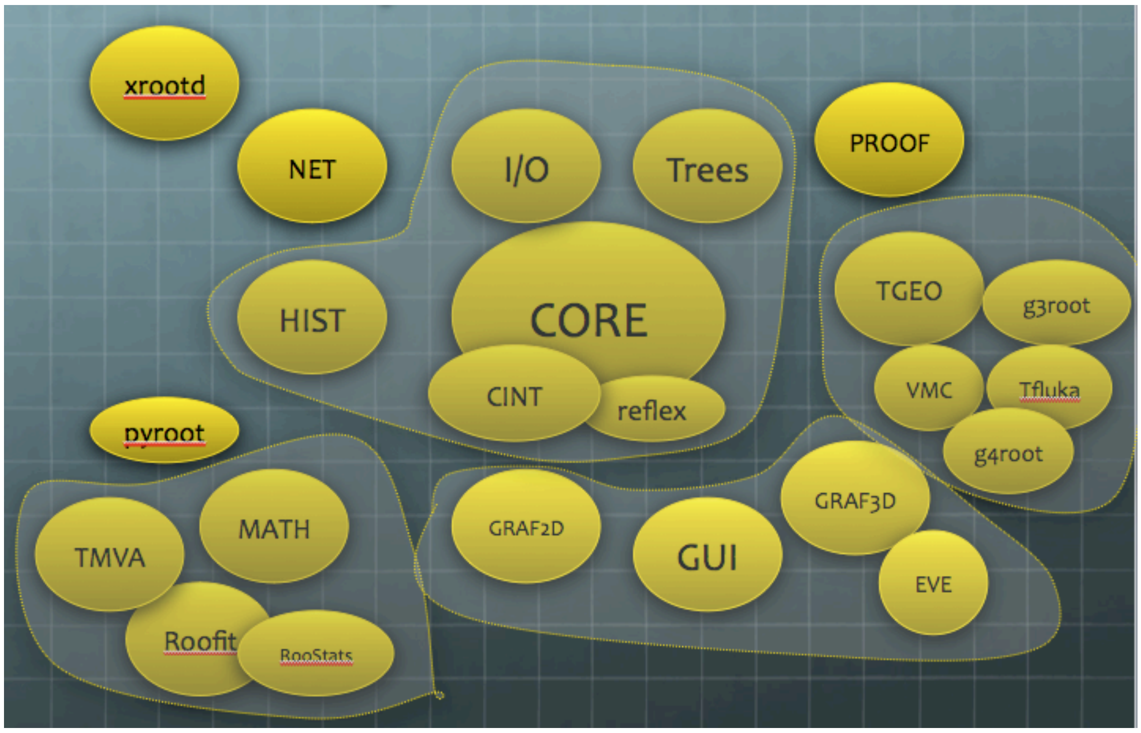

Figure 2: logical grouping of the different ROOT modules.

1.1 The CINT Interpreter and ACLIC

CINT [4] was originally developed as a C interpreter and then extended to support C++ features required by the I/O and graphics sub-systems. The success of CINT has been such that users have always pushed to support all features of the C++ language. The typical use is to start an analysis script with a few C++ statements to open a ROOT file and draw some histograms with more and more options. Then this script evolves as a more complex analysis system involving user classes, STL collections and complex code. At this point it becomes safer to, transparently via ACLIC, compile the script with the native compiler and link the code with the running executable module. ACLIC is the internal ROOT machinery that takes care of compiling and linking the script in a platform independent way. When executing "*.x myscript.C*+", ACLIC compiles *myscript.C* if it has been modified since the previous invocation. This process has proved to be very successful and it is our intention to simplify even more this task by replacing CINT by an LLVM [5] based compiler/interpreter with its Just-In-Time features. The move to LLVM has the advantage to provide support for the latest C++x0 version of C++. With LLVM, the interpreter is the compiler itself. The interpreter will be able to just-in-time compile scripts that otherwise would take far more time to execute, and this in a transparent way for users [8].

ROOT also provides the Python interface *PyROOT* that uses some of CINT features. This allows it to do dynamic call translation instead of relying on a fixed wrapper. Also provided is an interface to Ruby. Python and Ruby offer late binding and an easy to learn syntax. For a C++ framework, the major advantage of providing a C++ interpreter (e.g. compared with a Python interpreter) is the homogeneity of languages: users write compiled and interpreted code in the same language, they can transfer code or parts of it from the interpreted mode to the compiled mode without any transition.

1.2 The Math Libraries

The ROOT Math package consists of the following components:

- **MathCore**: a self-consistent minimal set of tools required for the basic numerical computing. It provides the major mathematical functions in the namespaces *ROOT::Math* and *TMath*, classes for random number generators, *TRandom*, class for complex numbers, *TComplex*, common interfaces for function evaluation and numerical algorithms. MathCore also provides basic implementations for numerical algorithms like integration and derivation. Furthermore, there are classes required for fitting the ROOT data objects (or any data set).
- **MathMore**: package incorporating advanced numerical functionality and dependent on external libraries like the GNU Scientific Library (GSL). It complements the MathCore library by providing a more complete sets of special mathematical functions and implementations of the numerical algorithms interfaces defined in MathCore using GSL.
- **Minimization and Fitting Libraries**: libraries required for numerical minimization and fitting. The minimization libraries include the numerical methods for solving the fitting problem by finding minimum of multi-dimensional function. The common interface for fitting is class *TVirtualFitter* and implemented by derived classes in the minimization and fitting libraries. The fitting in ROOT is being re-organized and new fitting classes are introduced in MathCore for providing the fitting functionality and the use of the minimization libraries via a new common interface (*ROOT::Math::Minimizer*). In detail the minimization libraries, implementing all the new and old minimization interface, include:
    - **Minuit**: library providing via a class *TMinuit* an implementation of the popular MINUIT [6] minimization package. In addition the library contains also an implementation of the linear fitter (class *TLinearFitter*), for solving linear least square fits.
    - **Minuit2**: new object-oriented implementation of MINUIT, with the same minimization algorithms (such as Migrad or Simplex). In addition it provides a new implementation of the Fumili algorithm, a specialized method for finding the minimum of standard least square or likelihood functions.
    - **Fumili**: library providing the implementation of the original Fumili fitting algorithm
- **Linear algebra**: two libraries are contained in ROOT for describing linear algebra matrices and vector classes:
    - **Matrix**: a general matrix package providing matrix *TMatrix* and vector *TVector* classes and the complete environment to perform linear algebra calculations, like equation solving and eigenvalue decompositions.

  - **SMatrix:** [7] a package optimized for high performances matrix and vector computations of small and fixed size. It is based on expression templates to achieve a high level of optimization.
- **Physics Vectors**: classes for describing vectors in 2, 3 and 4 dimensions (relativistic vectors) and their rotation and transformation algorithms. Two package exist in ROOT:
  - **Physics**: library with the *TVector3* and *TLorentzVector* classes.
  - **GenVector**: a new library providing generic class templates for modelling the vectors.
- **Unuran**: package with universal algorithms for generating non-uniform pseudo-random numbers, provides a number of classes of continuous or discrete distributions in one or multi-dimensions.
- **Foam**: multi-dimensional general purpose Monte Carlo event generator (and integrator). It generates randomly points (vectors) according to an arbitrary probability distribution in n dimensions.
- **FFTW**: a library with implementation of the fast Fourier transform (FFT) using the FFTW package.
- **MLP**: library with the neural network class, *TMultiLayerPerceptron* based on the NN algorithms.
- **Quadp**: optimization library with linear and quadratic programming methods. It is based on the Matrix package.

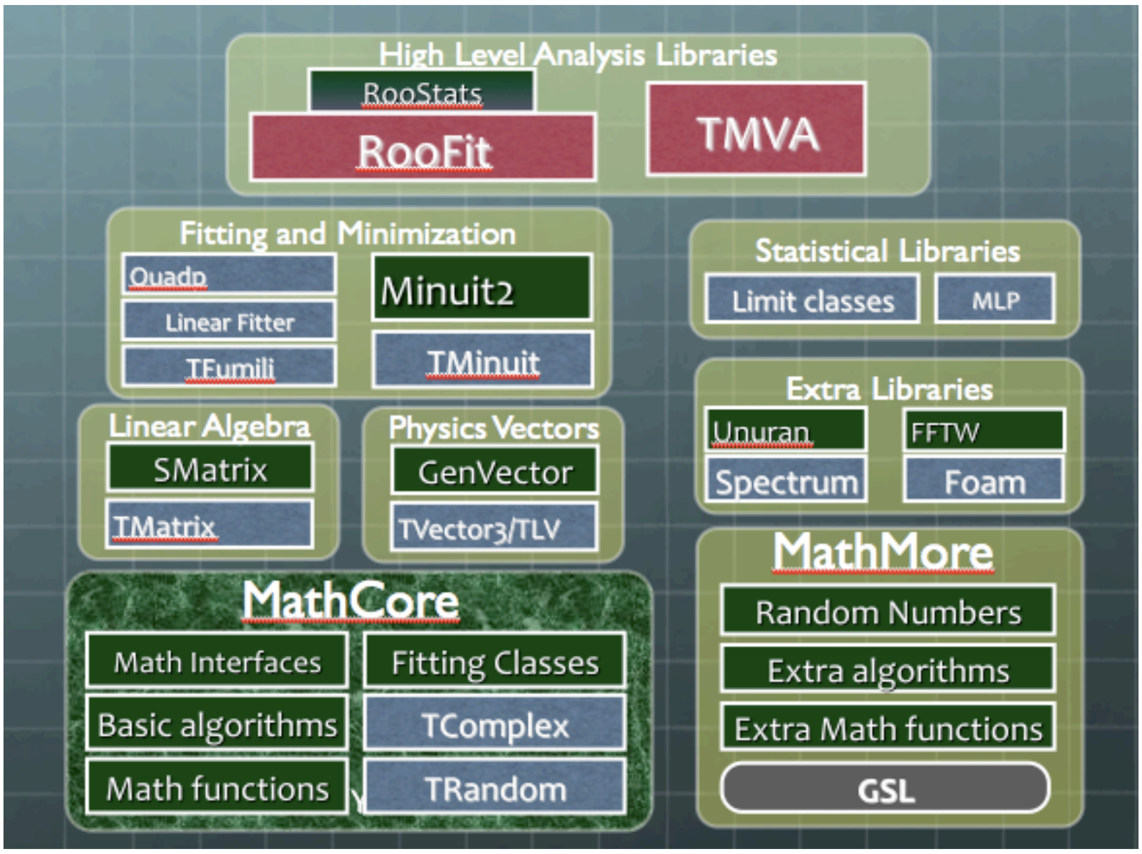

Figure 3: Hierarchy of ROOT math libraries.

2. 2-D and 3-D Graphics

The graphics classes are based on 10 years experience with PAW and 15 years of operation with ROOT itself. They have been designed to support the typical needs for presenting data in HEP. The system has been developed incrementally by adding zillions of tiny new features and options suggested by our large user community. These classes are designed to work equally well in an interactive environment as a batch environment where postscript and PDF files must be produced without running graphics systems like X11 or OpenGL. The classes produce high quality output pictures and provide interactive object editors.

There are several ways to render 3D graphics in ROOT. The preferred one uses the OpenGL graphics library, which is used in ROOT to display data using *lego* and *surface* plots and to render detector geometries. Work is in progress to also use it for 2D graphics and thus have a single, portable rendering interface for 2D and 3D screen graphics.

2.1 The Graphical User Interface

The ROOT Graphical User Interface (GUI) integrates typical GUI functionality with ROOT features, like exporting the GUI as C++ source, interpreting GUI code using CINT and CINT-based signal/slot event handling. The result is a flexible, cross-platform, GUI toolkit with a rich set of widgets and functionalities, including a GUI builder. The ROOT GUI builder provides tools for developing user interfaces based on the ROOT GUI classes. We also provide an interface with Qt. Although Qt is a popular and widely used system, maintaining the interface is quite expensive due to the many backward incompatible changes between the different Qt versions and their spotty deployment. Also history has shown that many systems considered as standards (*GKS* -> *Phigs* -> *Motif* -> *Qt*) have a relative short lifetime, at least with respect to the lifetime of HEP experiments.

Development is ongoing to extend the ROOT GUI and graphics to be fully OpenGL based. And we are investigating how to bring them to the browser via a JavaScript interface.

2.2 The Geometry and Event Display Packages

Geometry in 3D space is described in ROOT by means of basic solids that can be joined, intersected or subtracted to create more complex shapes. The possibility to visualize 3D objects is very important. ROOT implements its own scene-graph management library and rendering engine that provides advanced visualization features and real-time animations. OpenGL is used for actual rendering. Event display programs are an important application of 3D visualization. EVE, the event visualization environment of ROOT, uses extensively the ROOT data processing, GUI and OpenGL interfaces. EVE can serve as a framework for object management offering hierarchical data organization, object interaction and visualization via GUI and OpenGL representations and automatic creation of 2D projected views. On the other hand, it can serve as a toolkit satisfying most HEP requirements, allowing visualization of geometry, simulated and reconstructed data such as hits, clusters, tracks and calorimeter information. Special classes are available for visualization of raw-data and detector response.

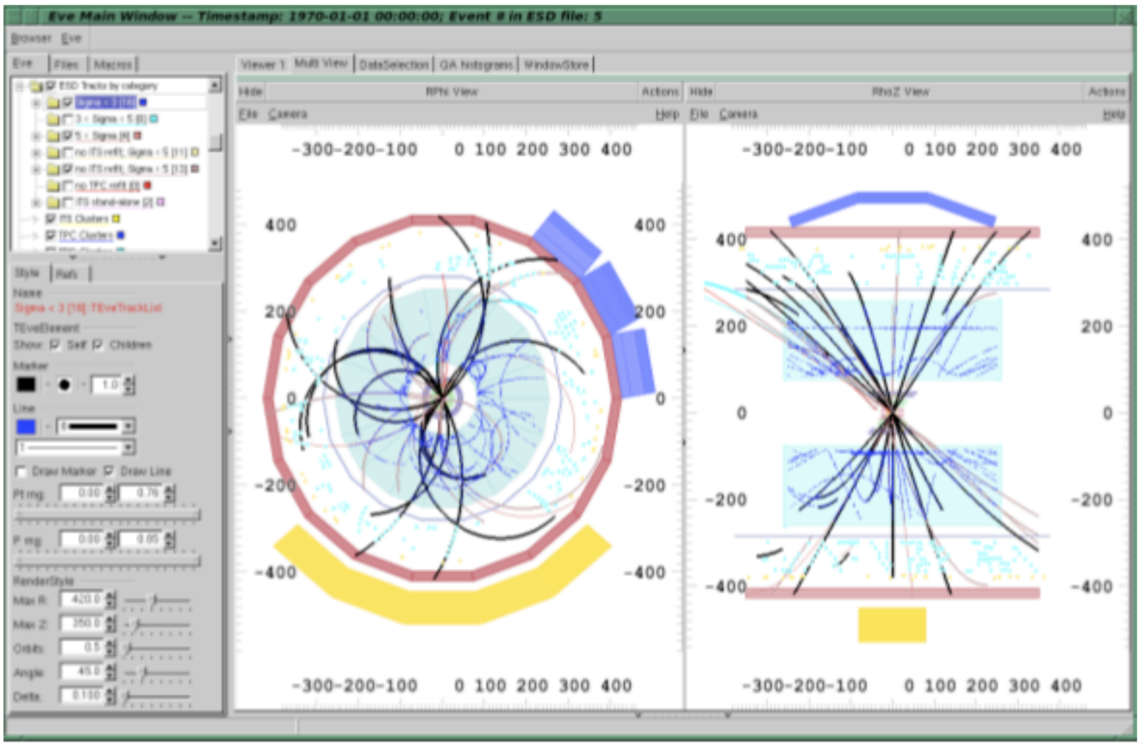
Eve Main Window — Timestamp: 1970-01-01 00:00:00; Event # in ESD file: 5
Browser Eve
Eve Files Macros
Viewer 1 Multi View DataSelection QA histograms WindowStore
RPhi View
RhoZ View
-300-200-100 0 100 200 300 400
400
200
0
-200
-400

Figure 4: EVE based event display, also showcasing the ROOT GUI.

2.3 I/O and Trees

A ROOT file is read and written by the class *TFile* and is designed to be write-once, read-often (while supporting deletion and extension of contained data). The content of a ROOT file is a simple binary stream consisting of variable length logical records, each one with a short header describing the record content. All data but the header is usually compressed to reduce the storage space and I/O bandwidth at the cost of slightly increased CPU time when reading and writing the files. The file consists of a content index, the list of type descriptions relevant for the file, and the actual data. Each data chunk is named and it can be retrieved by name. The *TFile* also supports hierarchical storage in nested directories. Typical file sizes range from a few kilobytes to several gigabytes. Files can be merged into new, larger files; this can be done recursively, i.e. merging also the collections themselves that are contained in the files, as long as they have the same name and are of the same type. Collections of files can also be merged into a zipped container; ROOT supports transparent unzipping of and navigation in this collection of files. The description of the classes stored in the file can be used to read the data even without the C++ class definition. One can thus write C++ objects using the definition from a user library, and read them back without the user library being available. Available reflection data is used to interactively browse a ROOT file using the *TBrowser* that can also expand and browse the content of all C++ objects, either from ROOT, STL containers, or user defined classes. ROOT files can be opened via the HTTP protocol, without any special server requirements. ROOT only asks for those parts of the file (using http content-range requests) that are actually required. This allows a low-latency, efficient remote browsing of ROOT files. In addition to the HTTP protocol, there are many more *TFile* plugins supporting different file access protocols.

A *TTree* is a container that is optimized for I/O and memory usage. A *TTree* consists of branches. Branches can contain complete objects of a given class or be split up into sub-branches containing individual data members of the original object. This is called splitting and can be done recursively till all sub-objects are split into branches only containing individual data members. Splitting can even transform containers into branches of the containee's data members. Splitting can be done automatically using the class dictionary information. Each branch stores its data in one or more associated buffers on disk. The optimal level of splitting depends on the typical future access patterns of a tree. If during analysis all data members of an object will be accessed then splitting will not be needed. Typical analyses access only a few data members; in this case splitting is highly beneficial. Branch-based storage is called vertical or column-wise storage (CWS), as opposed to horizontal or row-wise storage (RWS) as is usually found in RDBMS databases. In CWS, just like in RWS, a collection ("table") of similar objects ("rows") is assumed. However, in RWS all data members of an object are always read, while in CWS only the needed buffers (e.g. data members) are read. Splitting is an automated way to create these columns. CWS reduces the number of I/O operations and the amount of transferred data, because it reads only the needed parts of each object. All other members of the object keep the values defined by the class default constructor. When iterating through the collection, data members that need to be read are consecutive on the storage medium in the case of CWS. This allows block-wise reading of the data for several entries (rows) in one go, something massively favoured by all modern operating systems and storage media. Another advantage stems from the fact that ROOT compresses the data buffers using Huffman encoding, which benefits from seeing the same byte pattern more often, because the same data members usually have similar values (e.g. a particle's type ID). Because a *TTree* describes the objects it contains, one can read objects from a *TTree* even without their original class definition. The *TTree* can even generate a C++ header file representing the layout of the object's data as stored in the *TTree*. Combined with the power of the interpreter and ACLiC, this allows a smooth transition from stored binary data to C++ objects, even without C++ libraries. *TTrees* can also generate a *TSelector* skeleton, used to analyse the data.

3. How Did We Reach the Current Situation

ROOT is based on our previous experience with the development of CERNLIB and in particular packages like HBOOK, GEANT and PAW. The HBOOK system was the first large package in CERNLIB providing a powerful histograming and ntuple management system. Data could be saved in portable and compact HBOOK files and analyzed by interactive systems like PAW. The design of these systems was the result of a long interaction between the developers and the users. A short response time to implement a requested feature, the long term support and the service to help users were key features of these systems and it was essential to preserve this heritage when designing ROOT.

These CERNLIB packages and also the detector simulation package GEANT used a powerful data structure management system, ZEBRA, able to build complex data structures (in fact very similar to classes in an object-oriented system). ZEBRA provided a powerful way of writing these structures to files and back in a portable way.

During the development of PAW between 1985 and 1994 we realized the importance of efficient and flexible data structures for data storage in files. The success of row-wise ntuples pushed us to implement column-wise ntuples to support larger data sets where queries could be performed on a subset of the entries, subset of the columns or both. At the end of the PAW development, we also had a good idea of the user requirements for user interfaces, interpreters, graphics and access to large data sets. The Parallel Interactive Analysis Facility, PIAF, was a very important step to understand the issues with parallelism with the I/O sub system and the implications on the users analysis code. It was very important to understand the balance between CPU, disk I/O and network I/O.

3.1 Software Crisis in 1992-1994

It was clear in 1992 that new languages and new techniques would impact our Fortran-based systems. Many people thought that Fortran was the future. The MOOSE project investigating languages (C++, Eiffel) and techniques (UML, Rose, OO-design) was set-up in 1993. Many computer scientists became involved in the discussion, most of them with no experience with reality. Existing experience with large software systems was neglected as well as feedback coming from experienced physicists who were kept very busy by the design of the Large Hadron Collider. It took many years (of frustration) to move them from good old Fortran to the magic of OO and C++.

The direction suggested by MOOSE and other software *gurus* was to invest in commercial software systems for graphics and interactivity and move to commercial object-oriented database management systems. The assumption in 1995 was that OODBMS systems would dominate the world of databases in the near future. As a result a considerable investment was made in a commercial system, Objectivity, by the BaBar experiment at SLAC and the LHC experiments. Many people dreamed of a central database accessed worldwide at the bit level for writing and reading, without worrying about container structures, CPU and network performance; the technology helping to solve these problems. Most physicists trusted the *gurus* for implementing an efficient solution. The LHC Computing Board (LCB) monitored the progress. Unfortunately this committee relied too much on the same restricted set of *experts* who were assessing the progress with databases and graphics tools. Despite early signs of problems, these *experts* and the project referees persisted in their recommendations. *A posteriori* it is surprising to think that the Objectivity system could be trusted as a possible solution for object persistence in HEP.

4. The Initial Design and Development of ROOT

While the first ROOT prototype was a C++ version of PAW, we quickly realized that the key problem was to work out a general object persistence solution and that an OODBMS like Objectivity was not the solution. With ROOT, we continued the tradition started with CERNLIB of developing packages directly targeted to the most important needs of our experiments and with a long expected lifetime. For example, HBOOK has been in use for more than 36 years, Geant3 for more than 28 year and PAW for more than 25 years.

ROOT first focused on data analysis bringing the PAW experience to the object-oriented world, using an Open Source model of *release early, release often* to get contributions and feedback as soon as possible. The first ROOT prototype, released end 1995, was already used by the NA49 collaboration and quickly followed by many more adopters.

The development rule was primarily driven by user requests (bottom-up) rather than following the committee (top-down) approach. Another essential factor were the frequent User Workshops that helped shape and focus the ROOT development.

During the first 3 months of 1995, we prototyped several versions of the histograming classes, including a version based on templates. In October 1995 we presented a first prototype with histograms, functions, graphics (a la PAW). This prototype included a command line interface supporting very primitive calls to ROOT class member functions. This prototype generated a lot of interest and demonstrated that the ROOT development was going in the right direction and highlighted which components needed to be strengthened.

Rather than developing our own interpreter based on our very basic prototype, we decided to use an existing C/C++ interpreter, CINT, which had already been in use since 1991 and was developed by Masaharu Goto from HP Japan. We used CINT not only as an interpreter, but also as a parser for the ROOT header files to generate automatically a dictionary such that compiled class member functions could be called from C++ interpreted scripts and such that objects could be streamed to a buffer using automatically generated streamers. ROOT version 1 already included many more classes and much better graphics.

The I/O Streamer functions generated by *rootcint* were quite simple to understand and quite efficient. However, we realized soon that during the lifetime of an experiment and in the lifetime of ROOT itself, both the library classes and the user classes would be evolving and that at the same time the user would need, and expect to be able, to read files that were written with previous versions of ROOT. To support this evolution of the data models, we introduced class version numbers and the possibility for users to write their own custom streamers where they could take into account the class schema evolution. At the same time, we made *rootcint* more general such that it could parse many more classes, in particular non-ROOT classes developed by our experiment's early adopters.

We released version 2 with more features in the I/O, math and graphics sub-systems and more sophistication in Trees, our main container for event data.

In the fall of 1998, after extensive analysis of the existing options at the time, ROOT was selected by FNAL & RHIC as both their persistence and data analysis solution. This decision generated a big chock-wave in the establishment, as it was a clear signal that the officially supported top-down solutions were not measuring up to the need of experiments about to start running. This increase in the user base leads to a widening of the scope of ROOT.

4.1 More on the Design and Evolution of ROOT I/O

The ROOT streamer functions were very efficient and simple to understand. However the library containing the streamers was required in order to be able to read the data sets. While this requirement was not a problem when reading objects through an experiment framework, it made the sharing of ROOT files outside of these frameworks impossible. To solve this problem and to add more advanced support for schema evolution, we implemented an automatic system to stream data using the information in the dictionary instead of using the generated C++ streamer functions.

When using this dictionary based streaming, we also save to the ROOT file the meta-data information describing the classes of the objects stored such that ROOT files are now self-describing. Over time in the following years we continued to optimize this system:

- To support the full C++ language in general.
- To enhance run time performance of the streaming system.
- To support more sophisticated cases and more customization of the class schema evolution.

4.2 The Differences Between ROOT I/O and Objectivity

We list here the most important differences between ROOT I/O and Objectivity:

- In 99.99% cases people write a file once and read it many times. There is no need for an expensive locking process during reading.
- Files are self-describing and independent of the location where they are processed.
- Files are compressed and machine independent. Typical compression factors for LHC experiments are between 3 and 5.
- ROOT adds support for transient data members and does not stream them.
- ROOT can stream collections member-wise, optimizing both speed and disk space.
- Files can contain different class versions with an automatic class schema evolution.
- ROOT has a complete query and visualization system.
- Files can be processed in parallel with PROOF
- Files are processed orders of magnitude faster.
- ROOT has a powerful readahead and caching mechanism allowing reading files very efficiently across wide-area networks.

5. Conclusions

The ROOT development has been a great adventure lasting already for 15 years. Being the underdog during the early years was providing us with a lot of motivation to prove our vision. Currently ROOT has been adopted by basically the complete HEP community and the LHC experiments have used it to store and analyse the first LHC data. This is an absolute milestone. Currently, with the long running periods of the LHC, we are required to provide stable versions that have to be maintained during these long periods (up to 2 years). At the same time we have to avoid complacency and stay abreast of recent and new developments in the computing industry, like C++ language evolution, many-core CPU and GPU processing, the LLVM compiler suite, JavaScript browser side graphics rendering, etc. And in the meanwhile we have to continue improving the performance and functionality of the core system. We are not finished yet.

References

[1] R. Brun and F. Rademakers, *ROOT – An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nuclear Instruments and Methods in Physics Research A 389 (1997) 81-86. See also http://root.cern.ch.

[2] Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal,D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto,J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann,E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, M. Tadel, *ROOT - A C++ Framework for Petabyte Data Storage, Statistical Analysis and Visualization*, Computer Physics Communications 180 (2009)2499-2512.

[3] R. Brun, O. Couet, C. Vandoni, P. Zanarini, *PAW - A General purpose portable software tool for data analysis and presentation*, Computer Physics Communications 57 (1989) 432-437.

[4] M. Goto, *The CINT C/C++ Interpreter*, http://root.cern.ch/drupal/content/cint.

[5] C. Lattner, V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, Proceedings of the international symposium on Code generation and optimization, Palo-Alto (2004) 75.

[6] F. James, *MINUIT - Function Minimization and Error Analysis*. Reference Manual CERN Program Library Long Write-up D506, http://wwwasdoc.web.cern.ch/wwwasdoc/minuit/minmain.html.

[7] T. Glebe, *SMatrix - A high performance library for Vector/Matrix calculation and Vertexing*, HERA-B Software Note 01-134, December 2, 2003.

[8] Höcker et al., *TMVA - Toolkit for Multivariate Data Analysis*, CERN-OPEN-2007-007 (2007), arXiv:physics/0703039v4 http://tmva.sourceforge.net/.

[9] W. Verkerke and D. Kirkby, *The RooFit Toolkit for data modelling*, Proceedings to PHYSTAT05, http://roofit.sourceforge.net.

[10] A. Naumann and Ph. Canal, *The Role of Interpreters in High Performance Computing*, Proceedings of ACAT 2008, PoS(ACAT08)065, http://pos.sissa.it/archive/conferences/070/065/ACAT08_065.pdf.