ROOT – An Object Oriented Data Analysis Framework

Rene Brun [a], Fons Rademakers [b]

[a] *CERN, Geneva, Switzerland*

[b] *NIKHEF & Hewlett-Packard, Geneva, Switzerland*

The ROOT system is an Object Oriented framework for large scale data analysis. ROOT written in C++, contains, among others, an efficient hierarchical OO database, a C++ interpreter, advanced statistical analysis (multi dimensional histogramming, fitting, minimization, cluster finding algorithms) and visualization tools. The user interacts with ROOT via a graphical user interface, the command line or batch scripts. The command and scripting language is C++ (using the interpreter) and large scripts can be compiled and dynamically linked in. The OO database design has been optimized for parallel access (reading as well as writing) by multiple processes.

1 Introduction

Having had many years of experience in developing the interactive data analysis systems PAW[1] and PIAF[2] and the simulation package GEANT[3], we realized that the growth and maintainability of these products, written in FORTRAN and using some 20 year old libraries, had reached their limits. Although still very popular, these systems do not scale up to the challenges offered by the LHC, where the amount of data to be simulated and analyzed is a few orders of magnitude larger than anything seen before.

It became time to re-think our approach to large scale data analysis and simulation and at the same time we had to benefit from the progress made in computer science over the past 15 to 20 years. Especially in the area of **Object Oriented** design and development. Thus was born ROOT.

1 Tel.: +41-22-7672041, e-mail: Rene.Brun@cern.ch

2 Tel.: +41-22-7679248, e-mail: Fons.Rademakers@cern.ch

We started the ROOT project in the context of the NA49 experiment at CERN. NA49 generates an impressive amount of data, about 10 Terabytes of raw data per run. This data rate is of the same order of magnitude as the rates expected to be recorded by the LHC experiments. Therefore, NA49 is an ideal environment to develop and test the next generation data analysis tools and to study the problems related to the organization and analysis of such large amounts of data.

With ROOT we try to provide a basic framework that offers a common set of features and tools for domains, such as data analysis, data acquisition, event reconstruction, detector simulation and event generation.

Currently the emphasis of ROOT is on the data analysis domain but thanks to the approach of loosely coupled object oriented frameworks the system can easily be extended to other domains.

We believe that ROOT is an ideal environment to introduce physicists quickly to the new world of Objects and C++.

2 Architectural Overview

The backbone of the ROOT architecture is a layered class hierarchy with, currently, around 250 classes grouped in about 20 frameworks divided in 9 categories. This hierarchy is organized in a mostly single-rooted class library, that is, most of the classes inherit from a common base class `TObject`. While this organization is not very popular in C++, it has proven to be well suited for our needs (and indeed for almost all successful class libraries: Java[6], Smalltalk[7], MFC[8], BeOS[9], etc.). It enabled the implementation of some essential infrastructure inherited by all descendants of `TObject`. However, we also can have classes not inheriting from `TObject` when appropriate (e.g., classes that are used as built-in types, like `TString`).

2.1 The Class Categories

The basic ROOT classes contain the most low-level building blocks of ROOT. For example, the `TObject` class, which implements common behaviour for all ROOT classes. The class `TClass` and its helper classes that provide support for extended runtime type information. The storage manager `TStorage` which handles all memory allocation and de-allocation operations and performs basic error checking (memory overwrites, etc.). The class `TFile` which provides a hierachical sequential and direct access persistant object store. The operating system abstraction layer `TSystem` and the concrete OS interfaces `TUnixSystem`, `TWin32System` and `TMacSystem` concentrate all OS dependent behaviour, like file system access, dynamic loading and interprocess communication (IPC) for the three main platforms supported by ROOT.

The container classes provide general purpose data structure classes like, arrays, lists, sets, B-trees, maps, etc., which are heavily used in the implementation of ROOT itself.

The histogram and minimization classes offer advanced statistical data analysis features, like 1D, 2D and 3D histograming of short, long, float or double values, with fixed or variable bin sizes, profile histograms, data fitting, formula evaluation and minimization.

The Tree and Ntuple classes contain the tree system. The row-wise and column-wise Ntuples have been one of the major strengths of the PAW system. Trees extend the concept of Ntuples to all complex objects and data structures found on raw data tapes and DSTs. The idea is that the same data model, same language, same style of queries can be used on all data sets in an experiment. Trees are designed to support not only complex objects, but also a very large number of them in a large number of files. Ntuples are simple trees with one branch only.

The 2D graphics classes contain the low-level graphics primitives, like lines, arrows, rectangles, ellipses, text, etc., but also the higher level constructs like pads and canvases. They also handle basic style and attribute management.

The 3D graphics and detector geometry classes provide basic 3D graphics primitives, like 3D polylines and 3D polymarkers as well as higher level geometrical shapes (boxes, cones, polygons, tubes, etc.) which can be efficiently assembled into very complex detector geometries.

The MOTIF graphical user interface classes contain all the graphical and interactive components found in almost every user interface toolkit, such as buttons, windows, dialogs and menus. Similar classes have also been developed for Windows/NT and Windows 95.

The interactive interface classes and C++ interpreter allow the construction of interactive applications in which the user has to learn only one language, C++, to communicate with the program. The command language, macro language and programming language are all one and the same.

The documentation classes allow the creation of hyperized (in HTML format) C++ header and source files, inheritance trees, class indices, macro's and session transcripts. Thanks to this facility almost everything in the ROOT system can be automatically documented and cross-referenced.

2.2 *The* `TObject` *Class*

Most ROOT classes are derived from `TObject`. `TObject` defines protocols (abstract methods) for comparing objects, for object I/O, for graphics hit detection and for notification between objects, to name the most important ones.

The ROOT object I/O facility supports the transfer of arbitrarily complex polymorphic data structures from memory to a ROOT file and vice versa. This functionality is based on the abstract method `Streamer`, which is overridden in subclasses to stream an object's instance variables. Circular structures are linearized, and multiple references to the same object are restored properly. Storing pointers is implemented by an object table, which assigns a unique identifier to each transmitted object. This identifier can be transferred to other address spaces or to permanent storage.

Object I/O needs some information about the type of an object at runtime, because not only the state of an object but also its corresponding class type has to be transmitted. ROOT runtime support could provide enough information about an object's instance variables to implement the `Streamer` method generically in class `TObject`. However, we preferred the approach of a programmer selectively deciding which data members should be written to disk. Data members caching some state of an object that can easily be reconstructed in the `Streamer` method do not have to be transferred to disk. Another example is when variables can be compacted into short words or even single bits (booleans). To aid the programmer we provide a tool to generate automatically a default `Streamer` method.

The case of encountering an unknown class while reading back an object structure leads to the discussion of dynamic loading and linking. To handle this case gracefully, ROOT includes a mechanism to load a new class and link it to a running application. This dynamic linking support can be further used to extend a running system. In the case of the NA49 data analysis the library with the experiment specific classes is dynamically linked to the running interactive ROOT program.

The object I/O facility is also used as the standard format for transferring arbitrary data structures to other ROOT based applications running in other address spaces or on other machines. The transparent integration of dynamic linking into the object I/O mechanism allows the copying of instances of classes that are not known in the running application. This feature allows us to develop fairly easily advanced, web like, browsers that could operate on imported ROOT objects (for example, we could refit imported histograms without having to leave the browser).

Even with the upcoming runtime type identification (RTTI) extension for C++, the runtime system does not provide any information about the class structure, the instance variables or the member functions of an object. Consequently, an additional mechanism had to be introduced to gather this information, in order to support `InheritsFrom`, `Inspect` and `Dump` methods, the object I/O facility and the automatic documentation system. ROOT uses the approach of associating with each class (via a static pointer) a special object describing its structure. These descriptors are instances of the class `TClass` which is itself a subclass of `TObject`. `TClass` objects store the following information about a class:

- the name and title of a class
- the size of an instance in bytes
- its parent class(es)
- the names, types and descriptions of its instance variables
- the names and signatures of its member functions
- a source code reference to the definition and implementation part of the class
- the address of the class' object factory method used to create a new object

Because the C++ runtime system gives no access to type and structure information, the ROOT system uses a dictionary generator called ROOTCINT. ROOTCINT (a wrapper program around CINT, the C++ interpreter) parses the class header files and generates a dictionary (in the form of a C++ function). To link the ROOTCINT generated dictionary function to a class the programmer only has to add two preprocessor macros to his code. One macro, `ClassDef`, must be placed in the class definition file and the other macro, `ClassImp`, in the implementation file.

Besides as a dictionary generator, CINT is being used in the ROOT system as a command line interpreter and macro processor. Thanks to CINT the ROOT system can offer the user a single language (C++) interface.

3 The CINT C/C++ Interpreter

CINT is a C/C++ interpreter which is aimed at processing C/C++ scripts. Scripts are programs which perform specific tasks. Generally execution time is not critical, but rapid development is. Using an interpreter the compile and link cycle is dramatically reduced facilitating rapid development. CINT covers about 95% of ANSI C and 85% of C++. CINT, written in ANSI C (about

70000 loc), is solid enough to interpret itself and let the interpreted version execute a program. CINT makes C/C++ programming enjoyable even for part-time programmers.

CINT is developed by Masaharu Goto who is an R&D engineer in the mixed signal test departement of the Hachioji semiconductor test division of HP Japan. Masaharu is working closely with the ROOT team to integrate CINT seamlessly into ROOT and to further optimize CINT/ROOT.

The ROOT system embeds CINT to be able to execute C++ scripts and C++ command line input. CINT also provides ROOT with extensive RTTI capabilities.

4 The ROOT I/O System

One of the basic pillars of the ROOT system is its hierachical object database. The database is designed to be particularly efficient for objects frequently manipulated by physicists: histograms, ntuples, trees and events.

One could argue that this functionality can also be provided by a full fledged commercial Object Oriented Data Base Management System (OODBMS). We consider OODBMS's as potential candidates for the replacement of tools like HEPDB[4] or FATMEN[5], i.e. when locking and concurrent writing is required. But we do not believe that they provide a solution for the types of objects mentioned above. Why?

- Interactive computing is towards commodity desktop and notebook devices. They will be heavily used for histogram manipulation and data presentation. This should not require a special connection to a central data base or a license server (think of home computing).
- OODBMS's, by definition, are designed to store complete objects. Data clustering is organized around objects and containers of objects. They are not designed to access only a subset of the object attributes. We have demonstrated with the PAW column-wise Ntuples the usefulness of having access to single attributes. The ROOT Tree functionality cannot be provided in an efficient way by the current OODBMS's.
- OO data bases do not support on the fly data compression. We are designing experiments that will generate massive amounts of data. The cost of direct access devices for tens of terabytes may be a dominant factor in the cost of computing.
- Attribute range specification is not supported. A 4 byte integer cannot be saved as a single byte.
- The data bases companies are small and fragile. Will they survive in a few

years? The technology is not yet mature and compatibility between vendors is not guaranteed.
– We prefer not to discuss the question of the cost and the manpower resources necessary to support a commercial data base.

4.1 The Physical File Structure

A ROOT database file has a file header (less than 64 bytes) followed by several logical records of variable length. The first 4 bytes of each physical record are an integer holding the number of bytes in the record. If the number of bytes is negative, it identifies a deleted record that can be reused in a subsequent write operation. The following bytes contain all the information to uniquely identify a data block on the file.

The `TFile::Map` member function can be used to view the contents of a file by reading sequentially all the data blocks. The information stored by the ROOT output functions is always in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering). The redundancy in the logical record header can be used in case of file corruption or disk errors, to rebuild the original structure. Data in the logical records can be compressed or uncompressed, but the logical record header is never compressed. The first logical record on a file always contains the description of the top level directory of the file.

The logical records contain the following possible data:

– File/directory information.
– A standard object as written by the `TObject::Write` function. When the `Write` function is called, a `TKey` object is created. This `TKey` object is the logical record header.
– A `TTree` buffer. When a `TTree` buffer is written, a `TBasket` object is created. `TBasket` is derived from `TKey` and contains additional information specific to the `TTree` navigation logic.
– A user buffer. In the same way as for `TTree` objects, the user has the possibility to define a new class derived from `TKey` and optimize it for his objects.

Note that the concept of record length or block size has completely disappeared from the ROOT terminology. This simplifies considerably the logic of the system. It makes also simpler the implementation of memory mapping techniques.

4.2 The Logical File Structure

A ROOT file is like a Unix file system. It can contain directories and objects with an unlimited number of levels. Each directory has an associated list of keys, kept in memory until the file is closed. Finding an object on the file is done in two steps:

- Find the position of the key object using the key name.
- Get the position of the object on the file.

ROOT objects are always written in consecutive order on the file. By default, the directory description is written when the `TDirectory` constructor is invoked. However, if in a previous session, objects were deleted, the space released can be used by newly created objects. In this case, ROOT tries to find the best free block. The list of free blocks is a list supported by the `TFile` object. If a free block has a length that matches the length of the new object, the object is written in the free block and this free block is deleted from the list of free blocks. If not, the first free block bigger than the object is used. When a file is closed, the linked list of each directory is written to the file.

Thanks to this structure, a ROOT file can be read sequentially in case all objects need to be processed, or accessed ramdomly using the information in the `TKey` object. The `TTree` objects use a variant of the standard `TKeys` (`TBasket`). The `TBasket` keys are designed to address randomly a large quantity of objects (`TBranch` buffers) in very large files.

4.3 Support for Class/Schema Evolution

A ROOT file will in general be written with the same version of a class library. However, in the life time of a collaboration, the definition of many classes is likely to change frequently.

The problem is made even more complicated by the use of inheritance. Assume a class `D` with its base classes `C`, `B` and `A`. An object of class `D` must be identified by the four version numbers of its composing classes.

We have implemented in the ROOT files a versioning mechanism that guarantees that old files can always be read by new libraries. A likely case is the one of an analysis program looping on many data sets generated across the years with different class definitions.

The header files containing the class definition must include the macro:

```
ClassDef(Classname,VersionID),    e.g.   ClassDef(TLine,1)
```

The `ClassDef` macro is defined in one of the main ROOT include files which is automatically referenced by any include file using `TObject` derived classes. The second parameter of `ClassDef` is an integer representing the class version number. When the file is processed by the CINT Dictionary Generator the version information is saved in the dictionary data structure and is later available at execution time in the `Streamer()` I/O method.

4.4 The Input/Output Functions

An object is written to a file via the `TObject::Write()` method. This operation consists of the following steps:

– Creation of a support `TKey` object in the current directory.
– The `TKey` object creates a `TBuffer` object.
– The `TBuffer` object is filled via the `Streamer()` method of a class.
– If the file is compressed (default) a second buffer is created to hold the compressed buffer.
– Reservation of the corresponding space in the file by looking in the `TFree` list of free blocks of the file.
– The buffer is written to the file.

An object is read from the file into memory via the `TKey::Read()` method using the following sequence of operations:

– Search the key address by name in the current directory. The list of all the key names is contained in a hashlist.
– Create the buffer(s) necessary to read the object from file.
– The `TKey` object includes the name of the class of the object on the file. Using the class name, the pointer to the class definition (`TClass`) is obtained by looking into the linked list of class names supported by the top level `gROOT` object. The `TClass::New()` method is called. This function invokes the default constructor of the class.
– The object's `Streamer()` member function is called.

The prototype for the `Streamer()` function is automatically declared by the `ClassDef` macro. An Example of code for the `Streamer()` function is shown below for the ROOT class `TShape`. This code illustrates how the `Streamer()` function deals with base classes, including multiple inheritance. The two data members `fNumber` and `fMaterial` are integers. `fMaterial` is a pointer to the material definition for this shape. In the likely case that many different shapes will reference the same material, only one copy of the referenced material will be written. The code for this function is automatically generated by the CINT

Dictionary Generator.

```
void TShape::Streamer(TBuffer &b)
{
   // Stream a TShape object

   if (b.IsReading()) {
      Version_t v = b.ReadVers();
      TNamed::Streamer(b);
      TAttLine::Streamer(b);
      TAttFill::Streamer(b);
      b >> fNumber;
      b >> fVisibility;
      b >> fMaterial;
   } else {
      b.WriteVers(TShape::IsA());
      TNamed::Streamer(b);
      TAttLine::Streamer(b);
      TAttFill::Streamer(b);
      b << fNumber;
      b << fVisibility;
      b << fMaterial;
   }
}
```

4.5 Compression or No Compression?

By default, objects are compressed before being written to a file. The ROOT compression algorithm is based on derivatives of the well known gzip algorithm.

This algorithm supports up to 9 levels of compression. The default compression level is 1. This level is specified as a parameter in the `TFile` constructor or can be modified by the `TFile::SetCompressionLevel()` function. If the level is set to 0, no compression is done. The performance of the compression algorithm can be seen on an object by object basis by using the `TFile::Map()` function. Experience with this algorithm tends to indicate a compression factor of 1.3 for raw data files and around 2 on most DSTs files.

The time to uncompress a buffer is negligible compared to the compression time and is independent of the selected compression level. A ROOT file may contain objects written with different compression levels.

5 The ROOT Trees

For many years, the data flow model in HEP has been:

Raw Data Tapes $\longrightarrow$ Data Summary Tapes $\longrightarrow$ Mini/Micro DST's

The introduction of Ntuples in the PAW framework has proven to be very successful. Many experiments are using Ntuples as a convenient replacement for mini-DSTs or even DSTs.

The PAW Ntuples, however, were restricted to very simple data objects, collection of single variables or arrays.

With ROOT, we are introducing a new concept that we call Trees. Trees provide the functionality of the Ntuples and much more. The Tree architecture extends the concept of the Ntuple to all complex objects or data structures found in Raw Data tapes and DST's. The idea is that the same data model, same language, same style of queries can be used for all data sets in one experiment. Trees are designed to support not only complex objects, but also a very large number of them in a large number of files.

In a conventional DST, all data structures of one event were written in a contiguous area on the file. This model has been very successful and robust for sequential files and when the analysis program requires access to a large number of attributes of one event. On the other hand, this model was particularly inefficient when one had to iterate on a subset of the events or when only a small subset of the event attributes was used.

A Tree (class `TTree`) is made of branches. Each branch (class `TBranch`) is described by its leaves (class `TLeaf`). The leaves can be simple variables, structures, arrays or objects. An array may be of variable length, the length itself being a variable in the same branch or another branch. Branches will in general be objects. However, we thought important to also support variables and structures for these applications not yet converted to C++ and objects. A structure, for example, could be a simple C structure or a list of variables in a Fortran common block.

When the fruits of one branch (detector data) are ready to be picked, they are collected into baskets (class `TBasket`). When the baskets are full they go to the store, i.e. the file.

Each branch will go to a different buffer (basket). Some buffers will be written maybe after every event, whereas other buffers may be written only after a few hundred events. The different buffers can be organized to be written to the same file or to different files. This mechanism is also well suited for parallel architectures. Note that this scheme allows also insertion of a new branch at any time in an existing file or set of files.

Due to this data clustering scheme queries can be executed very efficiently. Queries executed on one or more variables or objects, cause only the branch buffers containing these variables to be read into memory.

Data are in general processed on different architectures with different memory sizes. In case the analysis is performed on a parallel architecture with a lot of memory, as many buffers as possible are kept in memory, maybe even all buffers.

The Tree data structure allows direct access to any event, to any branch and to any leaf even in the case of variable length structures.

6 Further Information

This paper describes only the fundamentals of the ROOT system. More detailed and up to date information can be found at:

`http://root.cern.ch/`

7 Acknowledgements

The authors of this article would like to thank the other ROOT team members, Nenad Buncic and Valery Fine and also Masaharu Goto for his fantastic help on the CINT/ROOT integration. Further we are grateful to the NA49 collaboration and the Hewlett-Packard Company for their support of our work on ROOT.

8 References

References

[1] PAW Users Guide, CERN Program Library Q121.

[2] PIAF Users Guide, CERN Program Library.

[3] GEANT Users Guide, CERN Program Library W5103.

[4] HEPDB Users Guide, CERN Program Library.

[5] FATMEN Users Guide, CERN Program Library.

[6] The Java Language Environment, J. Gosling, SUN Microsystems, 1995.

[7] SmallTalk80: The Interactive programming Environment, Adele Goldberg, Addison Wesley, 1984.

[8] `http://www.microsoft.com/`, Microsoft Corp.

[9] `http://www.be.com/`, Be Inc.