Uppsala Master's Theses in
Computer Science 304
2006-09-14
ISSN 1100-1836

Wrapping Persistent ROOT Framework Objects in an Object-Oriented Mediator System

Valentas Kurauskas, Matas Šileikis[1]

Information Technology
Computer Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

In this thesis we develop a wrapper for scientific data stored in ROOT files using Amos II, a functional DBMS. ROOT is an object-oriented framework for representing high-energy physics data. We investigate possible ways to represent ROOT C++ objects in a functional data model and propose a simple and fast mapping method. We define a schema to represent ROOT object storage containers, objects and their meta-data for Amos II users. The wrapper implements interface functions for data retrieval using the ROOT library. Amos II users can define SQL-like queries over data stored in ROOT files without dealing with levels of abstraction below Amos II. Athena is the common control framework for simulation, reconstruction, and analysis of scientific data collected in ATLAS experiment carried out in CERN. We investigate what is required to manage Analysis Object Data (AOD) files created using Athena. As a result, we show that our wrapper is capable of efficiently reading AOD files without having the ATLAS Athena framework installed.

Supervisor: Ruslan Fomkin
Examiner: prof. Tore Risch

[1] The authors can be contacted by email
valentas@gmail.com, matas.sileikis@gmail.com

Contents

1 Introduction

The Large Hadron Collider (LHC) is going to be the world's largest particle accelerator and collider located at CERN in Switzerland [12]. The LHC project aims to shed light on a range of cutting-edge modern physics theories such as the dark energy, the dark matter, extra dimensions, the super-symmetry, and give clues about unification of the four forces. LHC consists of a huge circular tunnel designed to collide counter-rotating beams of elementary particles. Particles are to be accelerated exploiting superconductivity and using electromagnets to reach a speed extremely close to the speed of light. To study what happens when the beams of that huge energy collide, five experiments will take part, each of them using detectors to measure, record and store physical events into digital data. This stage of experiments imposes a challenge for the best telecommunications and computing software and hardware as the collision data will contain enormous amounts of information.

Among the five experiments that construct detectors, ATLAS (A large Toroidal LHC ApparatuS) [14] is to date the most important for scientific computing. To help process streams of physics data produced by the ATLAS detector, CERN started up the so called Athena framework, a software package encompassing algorithms, filters, converters and other tools needed for physics research concerned with the data produced by LHC [13]. A final format of the collision event information is Analysis Object Data (AOD) units of compressed and minimized data organised as collections of objects. AOD data contain the part of all measurements essential and enough for particle physics analysis [14]. Section 3.2 gives more details about AOD. Much attention in CERN is paid for the Grid approach, which is a service for sharing computer power and data storage capacity over the Internet [20]. It is planned to be a common way to share and work with AOD data. Note that the concept of AOD also depends on the technology used to store data. The storage manager used behind Athena is the object-oriented, C++ based programming framework ROOT [7], aimed at solving the data analysis challenges of high-energy physics (section 3.3). Therefore accessing an AOD file can be achieved by directly accessing objects stored in a ROOT file. In ROOT, one of the key classes is *TTree*, a class describing a data structure for keeping collections of objects of the same type and accessing them rapidly. Higher level functionality, such as unique object and file identification, data hierarchy organisation, and transfer to the Grid for data storage to make it easily retrieved even without knowing physical storage details, is provided by the POOL (Pool of Persistent Objects for LHC) framework [18]. POOL plays a role of an intermediate layer between ROOT and Athena.

However, many data files are created separately, not following AOD format conventions, and using only ROOT. A vital requisite desired by scientists carrying out research on the particle data produced in CERN is ability to access experiment data in an appropriate programming environment. The only way to access such data currently popularized by CERN is using the Athena framework installed on a local cluster. Unfortunately, installing and using the Athena framework turns out to be a non-trivial and highly resource demanding task. Users who prefer a non-framework approach or for some reason have no access to any cluster with required CERN software have to look for other approaches. In this project we have developed the ROOT Object Wrapper (ROW) enabling database queries to scientific data stored in ROOT files (see description in section 3.3) This wrapper provides an easy and convenient way to browse and query ROOT object data files: retrieving file structure, listing hierarchy elements and stored data, getting data types and, most importantly, reading needed data using low level procedures to enable querying through a high-level query language [1, 19]. Such a system may prove useful since it provides easy access to

not only Analysis Object Data generated by LHC software but also to data stored using any other software based on ROOT as the storage manager. As long as the ROOT framework remains the technology used to store Analysis Object Data, all the functionality can easily be achieved by wrapping certain ROOT storage functions (more precisely, functions, for reading data from a ROOT file). Although the ROOT framework itself provides a graphical file browser, unlike ROW this browser is neither able to retrieve/list data nor formulate a specific query. In general all data retrieval functionality has to be manually programmed by the user. ROW aims to automatize this task as much as it is possible. Furthermore it concentrates on making the software compact to the highest possible extent, that is to use a minimal set of software making the installation and usage easier. As we show further in the report (section 6.1) this software volume is substantially smaller than that of the Athena framework.

ROW extends a functional object-oriented database management system, Amos II [1], to make it possible to query ROOT data files using the functional query language, AmosQL [1, 19]. Amos II (Active Mediator Object System) is a functional database management system that allows data from different data sources to be viewed uniformly as queries defined in AmosQL.Transparent access to different data sources is obtained by extensions of Amos II, so called *wrappers*, which convert data into Amos II data primitives – objects, types and functions, that can be processed and accessed by AmosQL queries. Amos II provides interface to foreign languages C, JAVA, and Lisp for wrappers implementations. Many wrappers providing access to common DBMS sources, wrapping various scientific data, music files, etc. have been created at UDBL [17].

We believe that the ROW approach is simple and attractive to common users. Instead of using the elaborate Athena or POOL frameworks to read data, we use directly the lowest level software, ROOT. Having our wrapper based on ROOT allows us to develop a simple, compact, easy to install, and easy to use cross platform software sufficient for querying data from arbitrary AOD files. Giving up the upper level software (Athena and POOL) means a disadvantage of losing certain functionality, such as universal object identification. Furthermore, a wide range of Athena's data analysis algorithms and tools are implemented as methods of objects that are callable (to date) only when the Athena framework is installed. However, in the case when only access to data stored in files is needed, the ability to easily query data from saved files outweighs the sacrificed functions. During the project we show that our ROOT Object Wrapper is capable of retrieving of all types of simulated (as the first real experiments are planned only in 2007) ATLAS detector data.

ROW uses ROOT's external call interface to access ROOT files through the external language interface of Amos II [3]. We introduce a new primitive type, *Struct*, to represent C++ structures in Amos II. In addition, we design a mapping from C++ primitive types and collections to basic Amos II types, making the wrapper able to represent in AmosQL most of the classes used for work with scientific data. For data retrieval we implement several basic functions, such as the function *get* to fetch data records one by one. These functions are implemented as foreign Amos II functions calling ROOT data retrieval functions and constructing Amos II *Struct* objects according to our designed policy. On top of the basic interface a meta-data model is implemented by defining an Amos II schema, consisting of entitity types such as ROOT container, ROOT type, etc. Furthermore, functions are defined to automatically fill and reuse this database each time a new ROOT file is accessed. We define routines to automatically collect the meta-data from each opened file as well as functions to navigate and extract particular data from retrieved objects using the meta-data. ROW is written in C++ and makes heavy use of the Amos II external call mechanism as the intermediate code between ROOT and Amos II. However, some non time-critical routines are implemented in AmosQL.

2 Example queries

To make a clearer picture of what the proposed software can be used for, let's start with two simple examples showing the basic capabilities of ROW. In this section we assume that the reader is aware of the basic Amos II concepts as well as with AmosQL language (if not, see [19] or Chapter 4). Here we skip most of the technical details, which can be found in the following chapters. The file we are going to analyse is of AOD format, that is, it has the same structure as the one which is to be actually used in the CERN's scientific research. The file is named *dc2.003007.evgen.A1_z_ee._00092.pool.root* and has been generated by a simulator using CERN software [22]. Here we view the objects stored in a file from the perspective of formal programming language, i.e. we do not consider the semantics of the classes.

The first action is to see all the containers stored in this file:

```
AODWrap 1> row_containers("dc2.003007.evgen.A1_z_ee._00092.pool.root");
...
<"","POOLContainer_DataHeader","POOLContainer_DataHeader",10001>
...
<"","POOLContainer_EventInfo","POOLContainer_EventInfo",10001>
...
<"","POOLContainer_McEventCollection","POOLContainer_McEventCollection",10001>
...
```

We choose to have a look inside the container *POOLContainer_EventInfo*. First we declare a new *RootContainer* instance and assign it a handle to a newly opened container:

```
AODWrap 1> declare RootContainer :c;
<#[OID 912 "ROOTCONTAINER"],AMOS_C>
AODWrap 1> set :c = openContainer("dc2.003007.evgen.A1_z_ee._00092.pool.root", "",
"POOLContainer_EventInfo");
….
6 types imported
```

OpenContainer creates several new meta-data objects describing all the classes encountered in this container: (in this case they are *EventInfo, EventID, EventType, TriggerInfo, …*):

```
AODWrap 2> select name(t) from RootType t where t = types(:c);
"vector<unsigned int>"
"vector<bool>"
"TriggerInfo"
"EventType"
"EventInfo"
"EventID"
```

The meta-object for *EventType*, for example, describes that structures of this type have two fields, one of which is a *Vector*, and the other is a *Charstring*. There is some other information

stored which is used for more complex structures. Here is a query for displaying all the members of the class (or type) *EventType*.

```
AODWrap 3> select attributes(x) from RootStructType x where name(x) = "EventType";

{"m_user_type",#[OID 991],1,1,0}
{"m_bit_mask",#[OID 1020],0,1,0}
```

A single result line contains a name of the member, a reference to an object describing it's type and several more values which will be explained later.

Where do all those types and names come from? In fact, the records we actually going to read are stored C++ objects of class *EventInfo*. The layout of this class is presented in Table 2.1.

EventInfo		
m_event_ID: EventID*	m_event_type: EventType*	m_trigger_info: TriggerInfo*
m_event_number: unsigned int m_run_number: unsigned int m_time_stamp: unsigned int	m_bit_mask: vector<bool> m_user_type: string	m_eventFilterInfo: vector<unsigned int> m_extendedLevel1ID: unsigned int m_level1TriggerType: unsigned int m_level2TriggerInfo: unsigned int

Table 2.1. Structure of obiects of class *EventInfo*.

Once we opened the container, we are ready to retrieve one entry from it:

```
AODWrap 4> declare Struct :s; /* Struct data type in AmosQL resembles struct type in C */

AODWrap 5> set :s = get(:c, 0); /* reads the first record in the opened container and
constructs an AmosQL object of type Struct according to predefined mapping */
```

The example starts with a procedure defined in the ROOT Object Wrapper, *openContainer()*. This procedure, together with the query function *get()* present the core functionality of the wrapper. The *get()* function constructs a "mystical" object [STRUCT #999] which is an Amos II structure (type *Struct*). Compare its representation (figure 2.1) with a C++ layout of an object.

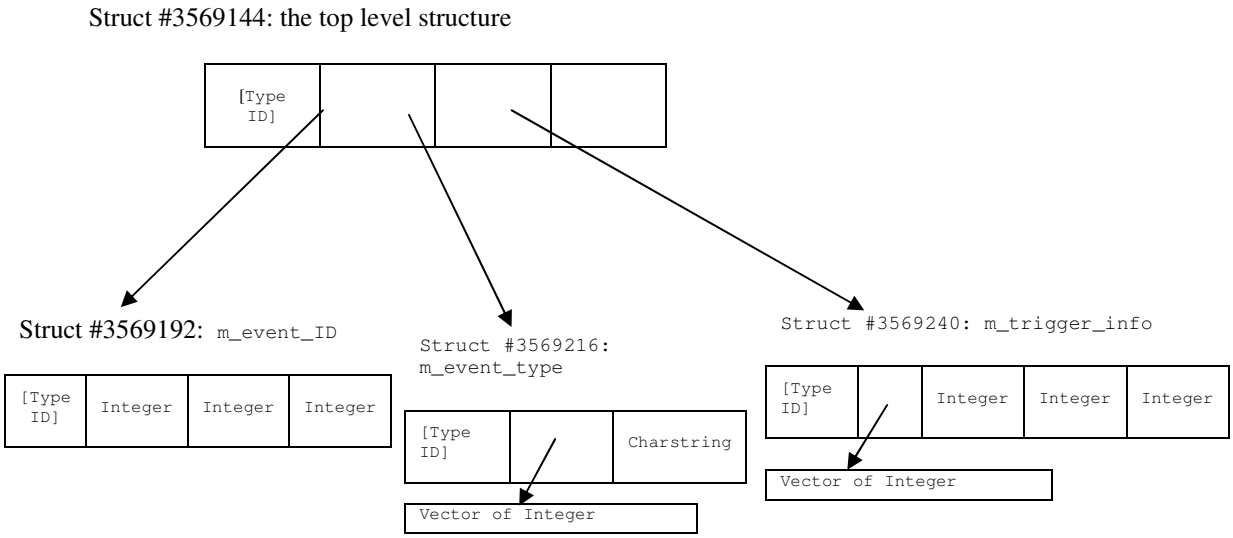


Figure 2.1. The *EventInfo* structure returned as a result of the function *get()*

Now we will try to access one more container:

```
AODWrap 5> declare RootContainer :c2;
<#[OID 925 "ROOTCONTAINER"],AMOS_C2>
AODWrap 5> set :c2 = openContainer("../data/dc2.003007.evgen.A1_z_ee._00092.pool.root",
"", "POOLContainer_McEventCollection");
20 types imported
```

At this point we skip several lines concerning some issues with circular structures. The subtleties are explained in Chapter 6. See Appendix A, Test E for a complete example using this function. To extract data member of type *HepMC::GenParticle* (describing a particle) nested in several levels within *McEventCollection* class (see figure 2.2) we can define a derived function which, given a *Struct* object (representing *McEventCollection* object) and Integer (index of the particle object we want to extract), returns a structure holding only data of the specified "particle" object.

```
AODWrap 9>
create function particle(Struct s, Integer i)->Struct
  as select getStructMember(
              getStructElement(
               getVectorMember(
                getStructElement(
                 getVectorMember(s, "m_pCont"), 0),
                "m_particle_barcodes"),
               i),
              "second");
#[OID 1045 "STRUCT.INTEGER.PARTICLE->STRUCT"]
```

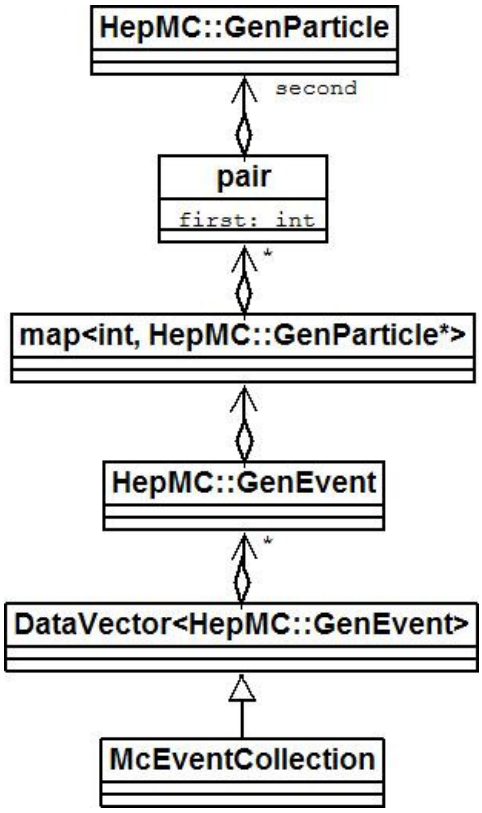


Figure 2.2 A particular path to *HepMC::GenParticle* data member within class *McEventCollection*

Next, we generate a thousand random integers from interval [0; 10000) and store them in the function *randints*:

```
AODWrap 10> create function randints(Integer)->Integer;
#[OID 1047 "INTEGER.RANDINTS->INTEGER"]

AODWrap 11> set randints(i) = r from Integer i, Integer r where i = iota (0, 999) and r =
rand(10000);
NIL
```

The following query selects records (actually, stored *McEventCollection* objects) according to random indices that have been just generated, extracts the *HepMC::GenParticle* member from each

of the latter objects, and extracts three coordinates of the particle represented by the *HepMC::GenParticle* member:

```
AODWrap 12> count( select getMember(mom, "dx"), getMember(mom, "dy"), getMember(mom,
"dz")
  from Struct s, Struct mom, Integer i, Integer j, Struct rec
  where i = iota(0, 1000) and
    rec = cast (get(:c2, randints(i)) as Struct) and
    s = particle(rec, j) and
    mom = cast (getMember(cast(getMember(s, "m_momentum") as Struct), "pp") as Struct)
);
```

Finally, we close all opened containers for memory cleanup.

```
AODWrap 13> for each RootContainer c closeContainer(c);
```

3 Background

In this section we discuss the most important technologies related to ROW. First, we briefly overview a field where ROW is claimed to be useful, i.e., scientific data management. Then, we describe a particular data format Analysis Object Data together with means for managing it. Afterwards we go deeper into the framework we used to implement it, ROOT. The last section is a very short description of wrapper/mediator approach but as the latter technology is crucial to ROW, we describe it to a greater extent in chapter 4.

3.1 Scientific data management

When the proton beams produced by the LHC interact in the center of the detector, a variety of different particles with a broad range of energies will be produced. Using elaborate and carefully designed inner components ATLAS will be able to detect high mass particles and measure their speed, positions, and momenta with a precision that was never available before. The trigger system, involving a cluster computer located near the detector, will then have to select the most interesting particle events out of 40 billion per second and store them for offline analysis. Event reconstruction will be performed by software which will involve operating on vast amount of data (hundreds of megabytes stored each second) and will require both extensive computational power which can be enabled using suitable methods for representation, storage and access of collected data [11]. A major interest for achieving this is in using the Grid.

To this extent CERN jointly with many international partners have established the Worldwide LHC Computing Grid (LCG) project which aims to ensure building and maintaining a data storage and analysis infrastructure for the entire energy physics community that will use LHC. LCG has many subareas, each of them devoted for specific tasks, such as 1) providing software for storage of tremendous amounts of data collected, 2) providing tools to service ATLAS detector in general and 3) creating a toolkit for all relevant calculations. Another important area of research in LCG is 4) simulating the detector data so that all the previously mentioned software could be tested and refined in advance [8]. Growing Grid capabilities and the amount of stored data and usage in scientific research requires higher level and easily applicable software that would prevent end users from putting too much effort in details of data access management. Data produced by the LHC experiments (or simulation of them) is only one example of excessively large amounts of data stored in nodes of the Grid infrastructure.

A number of frameworks and large software application projects are being implemented within and outside LCG. An example of these efforts is the Athena framework [13], which is a concrete implementation of functionality needed to develop physics applications using data generated by ATLAS. This framework in turn uses much of the previously developed ideas and architectures (Gaudi [23], GEANT [24], etc.) together with other software frameworks and applications developed by parallel LCG projects, e.g. ROOT and POOL [13]. See sections 3.2.3 and 3.3 for more about them.

An observation here is that the data files can be seen as databases of events. These databases are produced from one level of processing raw experiment data and are in general referred to as Analysis Object Data (AOD). The aim of ROW is to enable easy searching these databases using a database query language for scientific queries.

A particular scientific Grid project called POQSEC [2, 16] is in progress at the moment. The system relies on open source Grid middleware ARC [19] to access scientific data produced on the

Grid. The aim of POQSEC is to provide high performance scientific query execution over distributed and heterogeneous data sources accessed through ARC middleware. Wrapping and combination of data sources, query processing and optimization is performed by utilizing Amos II database management system supported and developed at UDBL. The work presented here provides a wrapper of AOD data for POQSEC to enable searches using the AmosQL query language.

3.2 Analysis Object Data (AOD)

As it was mentioned in the introduction during the experiments with LHC up to 1 petabyte of Particle Physics event data will be generated and stored each year. The problem being solved by projects in CERN is how to efficiently represent that data and how to make it available for using and searching in Grid environment.

3.2.1 Event Data Model

To provide an easy maintenance and coherence of collaborative experiments with large amounts of data generated by detector a C++ based Event Data Model (EDM) common for all the detector subsystems and groups was proposed at CERN. ATLAS EDM is designed to allow use of common software between online data processing and offline event reconstruction [14].

To facilitate the remote usage of event data, two types of datasets, or file formats[2], were introduced:

1) the *Event Summary Data* (ESD);
2) the *Analysis Object Data* (AOD).

ESD is designed to represent detailed output of the detector reconstruction and the data is produced directly from the raw data. Data in AOD format consists of summaries of each reconstructed event. It contains sufficient information for common analyses and can be generated from data in ESD format. The size of an event record in AOD is approximately 5 times less than the corresponding one in ESD. Thus AOD becomes the key format of storing the event data for scientific analysis.

3.2.2 Tracks, Events and Particles

The major aim of the *Event Data Model* is to share as much code as possible implementing the data obtained from various parts of the ATLAS subdetector systems, such as trackers and calorimeters. One of the most important concepts of this model is a common and efficient implementation of *track.* A track is a sequence of coordinates of a particle in a particular coordinate system**.** It is worth noticing that signals detected by various parts of detector must pass a number of reconstruction algorithms to finally be converted to meaningful and efficient representation.

The process of creating AOD files consists of:

- obtaining raw data from the detector
- reconstruction of particle tracks (to ESD files)
- preparation of tracks for efficient analysis (converting ESD files to AOD files)

[2] Later we will sometimes identify AOD and ESD concepts not as a format itself but as data stored in this format. The meaning should be clear from the context.

- including meta-information for events (event tagging)

The whole process also requires ability to store, or *persistify* the obtained objects.

The final representation in the actual implementation of AOD is object oriented event records. One of the currently used models of representing events and tracks is proposed in the HepMC package [9]. In HepMC each *collision event* is represented by a number of graphs, where each *vertex* is represented by a graph node and maintains a set of incoming and outgoing *particles* represented as edges of a graph. Multiple collisions are thus represented by a superposition of graphs. In addition to spatial coordinates many other parameters, such as particle charge, mass, lifetime, polarization, etc. can be stored in *event* objects.

Together with event objects, meta-data providing easier search among stored events also must be provided. Information stored as meta-data includes the time of the measured events, the type of their structural representation (as the classes for representing objects may evolve and change), and some universally unique identifiers. Furthermore, ATLAS EDM generally uses two types of storage: transient and persistent. See the following section for more about that.

3.2.3 Object persistence

Object persistence is another important issue tackled by physics projects related with Large Hadron Collider. The POOL project was established in CERN to help perform a largely collaborative analysis within the Grid environment and provide means for common object storage, identification and easier reuse. POOL aims to develop a set of service APIs for C++ programmers. These APIs introduce several abstraction layers of objects being persistified, including to navigate those layers, to retrieve the objects to virtual memory (as transient objects), and to store them on one of the types of storage (persistify). These layers provide a natural hierarchy for objects: containers, databases, catalogues, etc., altogether with associated meta-information for a quick access. As for the storage part, POOL builds on top of another framework, ROOT, i.e. POOL's storage collections are simply ROOT trees and databases are ROOT files. The navigational information is both stored in databases and in entities called catalogues, physically represented as XML files.

In this project we analysed many ways to work with AOD files. One of the first alternatives was to implement a wrapper using interfaces provided by POOL. Although the visions and ideas formulated for POOL are logical and seem to be useful, the projects turns out to face a number of difficult implementation issues.

POOL introduces distinction between physical objects, which can be either in ROOT containers called *TTrees* (section 3.3.3), in relational database records (not yet implemented), or in logical objects. It is claimed that one logical object can have several physical representations. A logical object is an object identified by a specific address and path (also introduced in POOL) and can be permanently stored in several places. Reading a logical object means implicitly connecting to one of its storage sites and transferring it to operating memory. This complicates the implementation of the framework.

Another issue in POOL complicating both programming and scientific analysis of object data is the abstraction layers described in the beginning of section. Event databases can no longer be stored in a single place: in addition to a physical data file there are some additional meta-data files. The layers and hierarchy itself requires user knowledge in order to be able to use it. Though those abstraction levels are not always necessary, still they must always be used, since POOL does not provide any other way to store objects. And lastly the introduced hierarchy requires defining an

addressing protocol, which can be difficult or unreasonable to learn for a user, especially if there is some other way to store the objects (for example, manually write them to the disk).

In general, until now the POOL project requires a considerable work to achieve its planned goals and currently provides only basic functionality. For now (version 2.4.3) POOL (a standalone version, independent of Athena) practically supports only one platform, namely CERN's SLC and a specific compiler [18, see Platforms and compilers]. It requires a dozen of dependencies to be installed and configured. This in turn requires either installing the same operating system as the one used in CERN [18, see Platforms and compilers, External Packages] or a greatly advanced knowledge and efforts to manually adapt the system. Clearly, it is not possible for POOL to fulfil the goals stated before starting it – to provide a common and easy access to objects within LCG, as 1) it is extremely difficult to install on most of the platforms 2) the interface requirements are formulated but implemented to a low extent 3) the interface is poorly documented. It is very likely that these issues make POOL available only to a narrow circle of users and the lack of testing and participation does not boost the development of framework.

However, a very useful service is POOL's use of ROOT to stream data from ROOT files. Such streaming can be done without using the POOL meta-data or the Athena system. This project provides ability to search these ROOT files ouside POOL by using a query language.

3.2.4 Storage of AOD data

In Figure 3.1 the data processing steps of Analysis Object Data are presented. It must be pointed out, that these steps do not use all the capabilities of Athena, POOL, or ROOT. Although it is only one of the many ways to produce AOD objects, it is currently the mostly used one. The first phase is the physical experiment phase, when software obtains raw detector's signals and applies algorithms to convert the detector input data to a particle track format and producing ESD files. The ESD files are further prepared to be saved as AOD using parts of Athena toolkit called *converters*. Then Athena calls its storage module to persistify resulting data. That is, it simply calls a built-in POOL application. POOL further adds its meta-data and calls ROOT libraries to finally store the objects into containers which are represented by ROOT *TTrees* [Section 0].

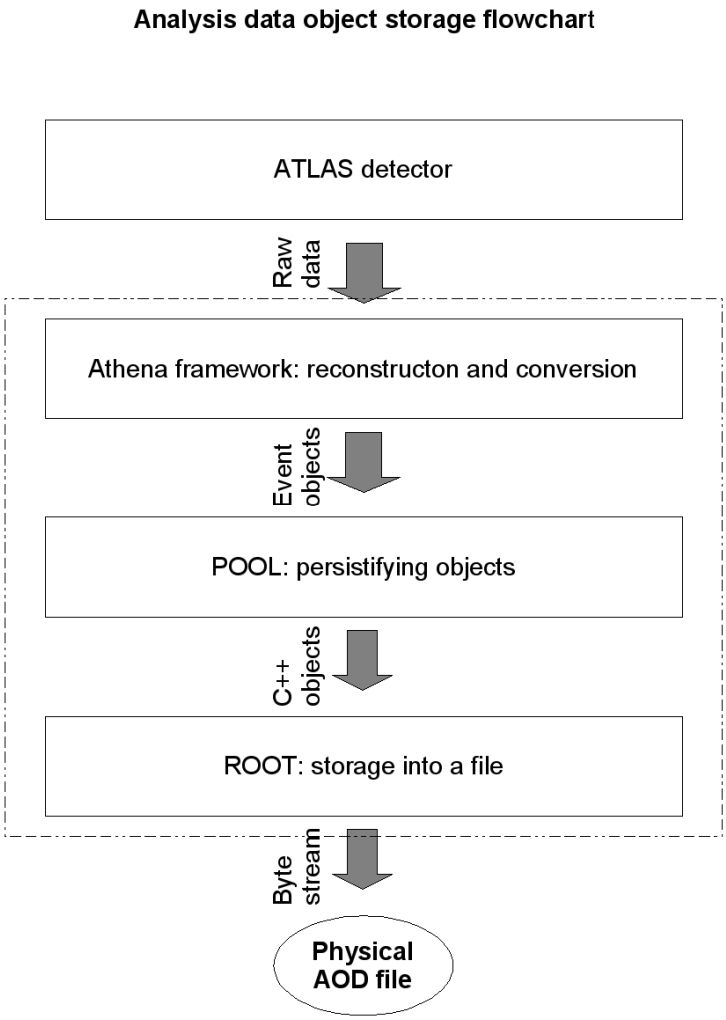


Figure 3.1. Generation of Analysis Object Data files.

Retrieving persistified objects back to analyse the stored data can be done in the same fashion, using Athena and calling special functions to open AOD files. However, retrieval of objects is different from their storing in that that AOD files basically need to be stored only once, after an experiment or a simulation, while they can be retrieved many times and by different users.

It turns out that there is another much faster and simpler way to access the AOD data, namely using only ROOT. A factor which can be exploited here is ROOT's ability to stream the objects even if the header, or class description, meta-data is not available through POOL. This ability is a part of the ROOT streamer and comes into play because ROOT aims to be able to store (almost) any kind of objects (simple class objects, ROOT class objects, even STL objects) self contained. Furthermore, as long as universal identifiers, cataloguing information, are not used in practice, it turns out that the main POOL's purpose is not more than calling ROOT interface functions to write the objects to certain containers (ROOT trees).

ROW utilizes the ability to directly access ROOT files containing AOD objects.

3.3 ROOT

ROOT is an object-oriented framework aimed at solving the data analysis challenges of high-energy physics. The framework was started in 1995 by René Brun and Fons Rademakers who had previously led other successful projects in CERN and developed a large, reliable, dynamic, and efficient system, which is today used in scientific analysis applications in many fields (e.g. particle physics, astronomy, biology, genetics, finance, etc.). ROOT contains components as a hierarchical object oriented database, a C++ interpreter (CINT), advanced statistical tools, visualization tools, a rich set of container classes, run-time object inspection capabilities, automatic HTML documentation generation, etc. [7].

Today the ROOT project is officially funded by CERN, has a solid group of developers, has a broad circle of users and testers, provides an elaborate user's guide [6], has good documentation, provides a ROOT-Talk Forum, supports many platforms (all Linux platforms, Windows), and is an excellent choice for physicists demanding a user-friendly programming environment

3.3.1 ROOT storage

Speaking only about storage component of ROOT, since 2001 important additional features were implemented making ROOT even more convenient and powerful.

First, foreign and emulated class support was implemented, allowing automatic or manual storing and retrieving any C++ objects from a file. That is, whenever an object of a new class is streamed into a file, the class (header) information is also stored to a file, making it self contained. Thus the ROOT streamer (i.e. the component responsible for writing objects as strings of bytes into file and vice versa) now does not need to use a C header of a class before the objects of that class are streamed. This is an essential improvement over the previous versions of ROOT where in order to read an object of particular class, the class header had to be compiled into a special runtime library called a *dictionary*. Recreating objects without a dictionary was impossible, which in turn required such systems as Athena and POOL to be installed. With the ROOT files now being self contained it is possible to access any ROOT file without consulting external meta-data. This is utilized in ROW.

Second, the *TRef* class was implemented which allowed fast and proper storing of pointer objects.

Lastly, a good C++ Standard Type Library (STL) support was provided and automatic streamers were implemented allowing easy and fast storage of virtually any kind of C++ collection into a file.

ROOT is an object oriented environment. Thus the storage component of ROOT (ROOT I/O) is based on class *TStreamer*. The streamer can store C++ objects derived from *TObject* (a common ancestor for all classes defined in ROOT) and objects of a class that has a dictionary loaded. For storing objects into a file, ROOT maintains a file header containing a description of the objects stored in the file. Every object being streamed is assigned a *key* string according to which it will later be retrieved. Sequential streaming provides many additional features and optimizations, such as object compression, fast seeking, object class evolution (objects of several versions of the same class may be stored), hierarchical structure of files (ROOT files may consist of the whole directory hierarchy), and chaining several files (objects stored in separate ROOT files may be linked using the *TChain* class).

3.3.2 TTree

The most important type of objects that can be stored into ROOT files are trees. Any other objects used in ROOT can be stored into a file by first placing it into a tree. However, the tree concept is different from that commonly used in computer science (digital, binary trees). It is designed to store a large number of objects of the same type (we will refer to a stored instance of a type as an *entry*), to quickly navigate, and to store and retrieve subsets of fields of objects and attributes. Therefore ROOT trees play the role of a database store. The authors of this project see the name for the *TTree* structure somewhat confusing, since a *TTree* is actually much more like an unnormalized relational database table than a simple tree data structure. A branch in a *TTree* can be imagined as a column or several grouped columns in a relational database table and a leaf as a column in the table. Saving objects in trees rather than streaming them directly into a file allows much quicker access and reduced disk space as there is no need to store meta-information for each stored object, but rather a single record for a whole group of objects of the same type.

TTree is a C++ class in ROOT, although it has many variations and extensions (*TNtuple*, *TChain*, etc.). *TTree* basically consists of one or more branches (*TBranch*) and each of the branches can be accessed independently of others. The lowest unsplittable element of a *TTree* is a leaf, *TLeaf*, which either holds one primitive data member (integer, float, string) or a whole binary object.

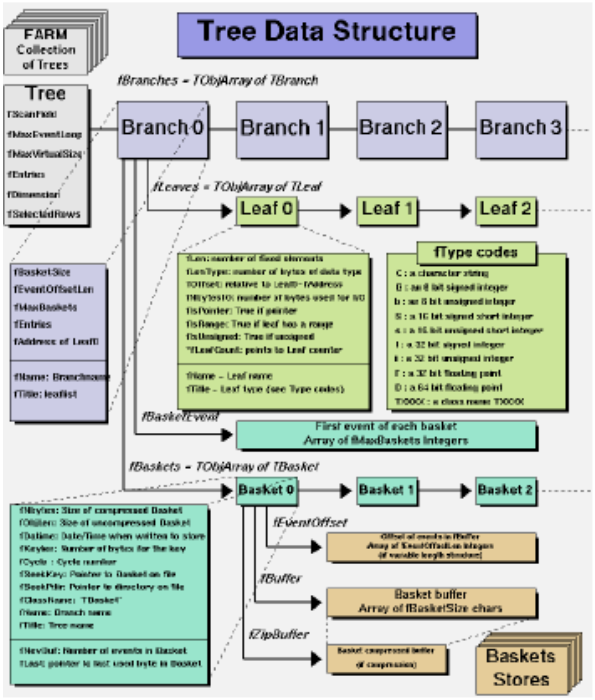


Figure 3.2, A structure of TTree

There are two basic ways to store data in a *TTree*.

The first way is storing tuples of primitives in a branch. That is we have explicit tuples of the values of fixed size and we want to store many of the samples with the same structure. The values can of course be both primitives and objects. An example could be a tuple of three doubles representing coordinates, `<dx, dy, dz>`. In this case each of the variables will be assigned one leaf in a tree and written together. When retrieving the data, the variables will be retrieved together. This case is equivalent to a relational database table with three columns (leaves) and a certain amount of rows (records). As coordinate values make sense only in triples, it is reasonable to store all of them in one branch. If on the other hand, we have subsets of values which are independent of each other, then those subsets can be stored in different branches of a tree. It means that when reading values back one is able to access only a relevant subset, and save the time for reading the other values. Using *TTrees* in this way is simple and quick and was taken by a number of applications. A wrapper for Amos II developed by J. Tysklind [5] wrapped exactly the trees produced by this method. However, this approach (writing tuples of primitives to a tree) is becoming less popular because of the following disadvantage: for each new class of objects the tree creation, data writing and retrieving procedures must be programmed manually. Given the new capabilities of ROOT of introspecting its classes, an alternative approach is natural: if the system has its object layout described it could automatically create branches and assign data members or read them back.

The second way is storing objects in a branch. We store an arbitrary number of objects of the same kind of branch. Although, one of the options is to save each object as a bunch of bytes, just like as it would be done in the first case, ROOT allows automatic splitting of objects that are being written. That is, if an object contains aggregated object members inside, a sub-branch (physically it is also at the same level as the top-level branch, the 'splitting' is only logical) can be automatically created for each of those aggregated members; the same can be done for their child members, etc. How deep the objects are expanded can be controlled by setting the split level property for a *TBranch*. If the split level is more than 1, the members of a stored object can be retrieved separately from each other, thus ensuring a fast performance. The automatic splitting also handles writing collections, but currently only of the types defined in ROOT. The remaining collections, such as STL vectors are usually not split, i.e. streamed as binary objects.

If data is written to a *TTree* as objects the easiest way to read it back is to read it also as objects and this is supported by ROOT's streamer. If the split level is high, primitive members can be accessed one by one, otherwise entire top level objects must be retrieved, and the programmer has to set up object additional routines to provide the application user an ability to extract required members from those objects.

Storing objects, not tuples of values in a branch was chosen in the implementation of the POOL project [see section 3.2.4]. As long as we take the object oriented approach to represent Particle Physics entities, such as particles, tracks, vertices and events, we also store them as objects, not as tuples of preselected values. Furthermore, because ROOT files are self contained, ROOT is able to read any object from the file, not only the ones which are defined in the program (see the following section). This makes it possible to read object data members from such files as AOD without using any higher level frameworks, including Athena and POOL.

3.3.3 Reflection

Reflection is the ability of programming language to introspect its data structures and interact with them at runtime without prior knowledge [6]. ROOT provides reflection for C++ with a Reflex package which replaced the older techniques used before ROOT version 5.08. To be able to use this reflection for a particular class, a runtime dictionary library must be loaded for that class. For

standard ROOT classes, dictionary libraries are provided within the framework itself. For own defined classes user must use tool programs to generate dictionaries.

A *dictionary* represents a static C++ class header file which is loaded at runtime. Using dictionary, objects of the class it describes can be created and manipulated, including retrieval of an object from a ROOT file or tree.

However ROOT provides an alternative option for reading of objects that were streamed into a ROOT file – dictionary-less object creation. Starting from version 5.08 ROOT is able to recreate an object from a file even in the case that object is not defined previously and its class does not have a dictionary loaded. To ensure this, for each class and any of its super-classes of objects stored in a file, ROOT stores information in the file header required to recreate and read the object back. When the main reading routine is called, object is automatically recreated and pointer to it returned. If the dictionary for an object is loaded, this pointer can be directly casted to the object of a correct class. Streamer information (offsets and member object names and types) can be used to navigate through the fields of an object and reconstruct the information stored inside. In other words, all objects which are stored to a ROOT file have their layout described in the same file and can be read without any other information. The objects created without *dictionary* are referred to as *emulated objects* in ROOT.

If, additionally, the dictionary for the class is provided, the class methods can also be called Special handling and navigation is provided for navigating collections and arrays using a ROOT class *TVirtualCollectionProxy.* A need to be able to navigate the objects using streamer offset information arises as it is the only way to reconstruct those stored objects in a tree which were not split[3]. (if object was completely split each primitive member would be stored in a separate branch and simply names and types of branches could be used). ROW is directly based on retrieving data using ROOT's reflection mechanisms. That is, before reading any object it first accesses the metadata (such as the layout of members). See the implementation part (section 6) for more about that.

So far the Reflex package does not provide ability to also store and call object methods from a file. However this ability is planned to be implemented in future [personal communication].

3.3.4 Why access AOD using ROOT?

As it was described before, the most natural way to access AOD data files is to use the same tools as were used to store it. That is, use the programming environment and functions of ATLAS' Athena framework with built-in POOL. An obvious disadvantage at this point is the size and complexity of the framework. Having hundreds of megabytes of source code and a variety of required software, Athena becomes not a trivial program to install. The demand of resources and evan a specific platform makes it reasonable to be installed only on servers in large networks or clusters. This may not be appropriate for all of the users, since it is often only possible to work on a disconnected computer (laptop) or in a system which does not belong to a cluster where Athena is installed. Another disadvantage, already mentioned in Section 3.2.3, is a demand for knowing most of the protocols used in all the layers (Athena, POOL, ROOT) to manage the files, even if these protocols provide only redundant functionality.

Together with our work we propose an approach which is much simpler and much more attractive to common users. That is, we claim that using only the framework of the lowest storage

[3] The split level is set when creating a tree. I.e. members of member of an object are assigned separate branches only when the split level is more than 1. Typically the split level in AOD files is high but lower branches are not split because they contain such structures as STL vectors of objects and splitting is not implemented for them..

level, ROOT, is enough to carry out certain kind of research with particle physics data, i.e., search for certain kind of data. A circumstance which enables us to do this is the way object collections are persistified (i. e., stored to a physical memory) in POOL as ROOT files. Containers are simply ROOT *TTrees* of the second type where a branch consists of a number of object records. Thus because the POOL implementation does not use any additional processing when storing objects into files than that provided by ROOT we can use ROOT to browse and retrieve stored objects.

It is true that accessing data that has some hierarchical wrapping from a level other than the top one makes us sacrifice certain functionality which was known only for the higher levels. For AOD data we lose a place of collections objects in the directory structure. Collections in POOL must additionally be stored in catalogues, which are implemented creating records in an additional XML file. Also POOL aims to provide meta-data search in its catalogues or collections. This functionality of course can not be accessed by only working with a physical ROOT file, as the additional information is stored outside. So is, for example, the UUID (Universally Unique ID) for each of the files. If a user uses ROW, he or she must know exact name of the file where the data is stored and have the file accessible for the wrapper. On the other hand, file information is stored as meta-data in an Amos II database and can be searched. Because physicists will want to work with a limited collection of files there is often no need to store the whole directory hierarchy of all data files.

A big advantage of reading AOD directly using ROOT versus reading using Athena is simplicity and transparency. We no longer need to have POOL or Athena installed which is very complicated, we need only ROOT which is, like it was noticed in the chapter overview, much easier to install and much more user-friendly. As long as ROW deals only with reading and searching data, class dictionaries are also not necessary.

3.4 Wrapper and Mediators

Scientific data is distributed among peers of the Grid and sometimes needs to be combined from data sources of different kind. Therefore a reasonable solution for performing queries over scientific data is to apply the mediator/wrapper approach [18]. A *mediator* is a network peer that is seen by other mediators and applications as a set of data views defined in a common data model (CDM). These views access data sources by so called *wrappers*. Each kind of data source has a corresponding wrapper that translates its data into the CDM. A mediator system enables integration of several heterogeneous and distributed wrapped data sources. The Amos II system provides mediator and wrapper functionality.

4 The Amos II system

In this chapter we will briefly describe features of the Amos II system that are related to ROW and explain certain aspects in more detail to make it possible to understand the architecture and implementation of ROW. See [1, 2, 3, 4, 17] for more about the system and its architecture.

4.1 Overview of Amos II features

An Amos II system may have arbitrary number of wrappers that convert external data sources into the Amos II CDM and provides means for executing queries upon them.

Wrappers may be written in C, JAVA and Lisp, as Amos II provides a so called *call-out* interface for these languages allowing wrapper implementations. ROW is a wrapper is written in C++ and implements an Amos II call-out interface.

The *call-in* interface [3, 4] defines calls of opposite direction, i. e., how programs can access Amos II, and execute AmosQL queries through their mediator peers. The call-in interface is also used by ROW because, as you will see in chapter 6, meta-data of ROOT files is stored in the Amos II database and needs to be accessed by ROOT Object Wrapper.

4.2 Functional data model

Amos II employs a functional data model that represents all data as *objects, types and functions.* To simplify understanding we will provide corresponding object-oriented concepts.

4.2.1 Types

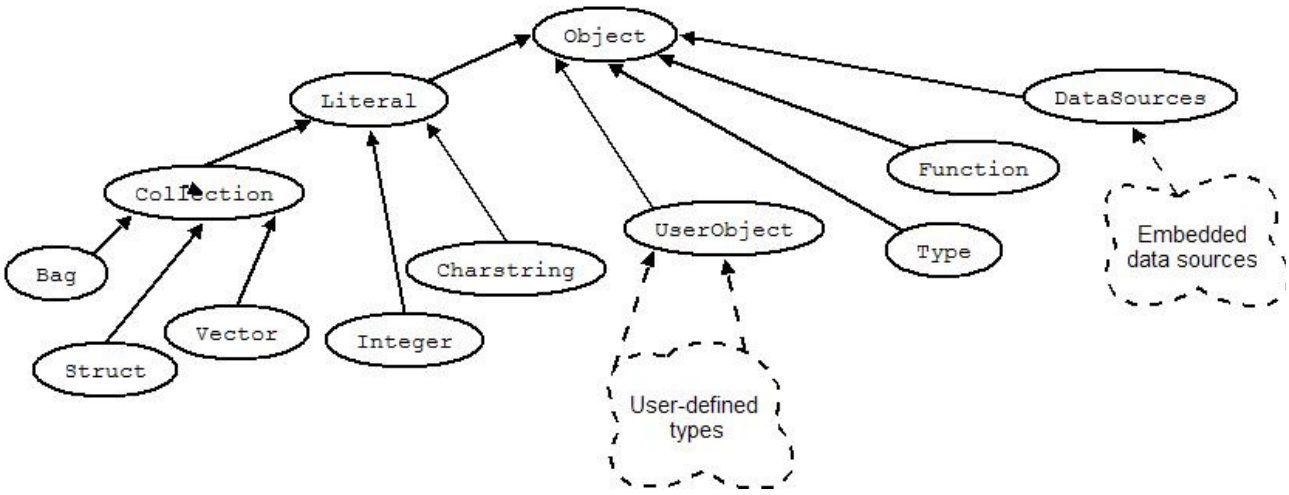


Figure 4.1. Types in Amos II.

An Amos II *type* corresponds to a class. The system provides a hierarchy of built-in types displayed in figure 4.1. User can define new types and create objects for them. User defined types

automatically become derived from the type *UserObject*. User can also create types hierarchies under earlier defined user types.

4.2.2 Objects

Objects correspond to object in the OO paradigm and are instances of some type. In Amos II anything is an object, including types and functions themselves. Objects are of two basic types:

- *Surrogate* objects have associated numeric key identifiers (OIDs) that are unique within the local Amos database. All objects of user defined types are surrogates. The OIDs are managed automatically by the system. Surrogate objects have to be explicitly deleted by the user to be removed from the database when they are no longer needed.
- *Literal* objects don't have associated OIDs and are automatically deleted by a garbage collector whenever they are no longer referenced from other database objects. They are more light-weight than surrogate objects. Examples of literals are *Number*, *Charstring*. Literal subtypes that are special are *Collection* objects. In case of ROW, the task of the wrapper is to represent data stored in AOD/ROOT files as literal objects so that huge amounts of data can be efficiently streamed through the database. ROW uses three kinds of collection objects:
  - *Bag* – set of any objects
  - *Vector* – zero-based indexed array of objects
  - *Struct* – a special collection representing records (C structs) containing:
    - A reference to an Amos II *type* object. It is up to developer to implement type representation in Amos II and assign a type object to an instance of *Struct*.
    - An indexed array of object references by the *Struct*.

4.2.3 Functions

A *function* represents a property of some object, a mapping between objects of different types, or a computation over objects. They can represent not only one-to-one functions (having mathematical meaning) but also mapping to several objects (thus representing one-to-many and many-to-many relationships). They can be classified as:

- *Stored functions* to represent properties of objects (attributes) locally stored in an Amos II database. Stored functions correspond to attributes in object-oriented databases and tables in relational databases.
- *Derived functions* are defined by a single query statement (SQL-like select statement) and therefore can be defined on other Amos II functions. Derived functions cannot have side effects. Derived functions correspond to side-effect free methods in object-oriented models and views in relational databases.
- *Stored procedures* that are defined using procedural sublanguage of AmosQL and are meant to be means for computation or correspond to methods with side-effects.
- *Foreign functions* are defined in a programming language C/C++, Java, or Lisp. They provide ways of calling external systems from Amos II. Wrappers are defined using foreign functions.

Foreign functions can be defined as being *multidirectional*, meaning that both the function and ints inverses are defined. For example, the following function is multidirectional:

```
create function f(TypeA a)->TypeB bas multidirectional
("bf" foreign "A_to_B" cost {100, 1})
("fb" foreign "B_to_A" cost {200, 1})
("ff" foreign "scan_A_B" cost scan_cost)
```

The above definition means that mapping from A to B $f(a): A \rightarrow B$ is performed by foreign function `A_to_B` (*binding pattern* "bf" means that first argument *a* is bound, and the second argument *b* is free), backward mapping $f^{-1}(b): B \rightarrow A$ is meant to be performed by foreign function `B_to_A`, and foreign function `scan_A_B` should return all pairs $(a, f(a))$ ("ff" means that both arguments are free).

The cost specification provides estimates for the Amos II query optimizer about function execution time and size of the result returned. This can be specified as constant in AmosQL (first two cases) or be returned by external cost function, `scan_cost`, in this case.

This kind of function definition allows Amos II efficiently process queries like

```
select a from TypeA a, TypeB b where f(a) = b
```

and to choose efficient function execution depending on the cost. In this wrapper the foreign functions *row_get* and *row_scan* implement the multidirectional function *get* (see chapter 6.2).

4.3 Data storage

The Amos II DBMS uses a main memory database storage manager AStorage. All data in an Amos II database is stored in a *database image* managed by this manager. AStorage is responsible for allocation and deallocation of physical objects inside the database image. A physical object is a C structure handled by the storage manager. Amos II objects (logical objects) described previously are of a higher abstraction layer built on top of AStorage physical object layer.

The C implementor has the choice of allocating data *persistently* inside the database image by using a set of primitives provided by the storage manager (also refered to as *storage interface* in this paper). Persistency in this case means that data allocated inside the database image can be saved on disk (for example, by issuing AmosQL statement 'save') and later restored.

The C/C++ programmer can define own persistent data structures by using a set of storage manager primitives. By employing this feature ROW creates both literal and surrogate Amos II objects.

5 System Architecture

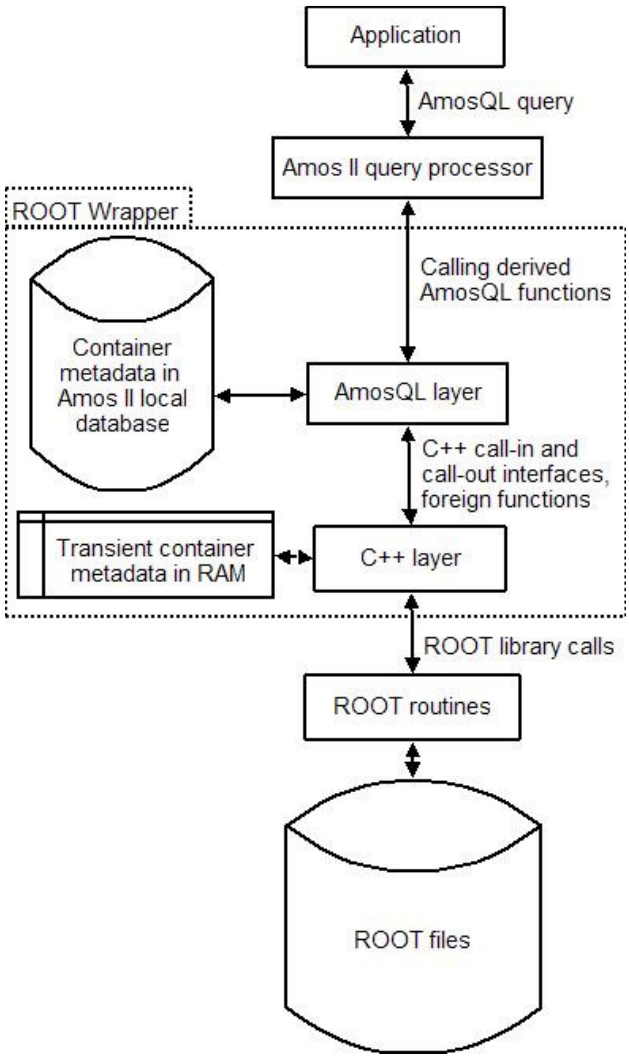
5.1 Layers of ROOT Object Wrapper

An Amos II wrapper consists of two abstraction layers one of which is defined in a foreign language, as the external source provides interface to some external language, e.g. C++, and the other is defined in AmosQL, as the source must be queryable by Amos II users.

The ROOT Object Wrapper has the following abstraction layers:

- The lower one is written in C++ and implements an external C interface of Amos II. It includes:
  - *Transient container meta-data storage*, a structure to hold transient meta-data describing the external ROOT data storage being wrapped. The data in this structure is not used directly by Amos II user but is only for better performance.
  - Routines employing ROOT framework to manage data stored in ROOT files.
  - Routines in ROW using the call-in interface to access the Amos II database to create transient and persistent objects.
  - Functions implementing ROW functionality through the call-out interface.
  - Caching and optimization components.
- The upper layer is written in AmosQL[4] It includes
  - Definitions of external functions provided by the lower layer.
  - Definition of meta-data schema (see next section).
  - Meta-data management procedures.
  - Definition of literal type *Struct* (see next section) and derived functions for more convenient management of data retrieved by external functions.

[4] Only the type *Struct* is defined in Lisp.



5.2 Meta-data

5.2.1 Structure of objects stored in AOD files

As it was explained in section 3.3, our task was to extract C++ objects stored in ROOT files in a form which is convenient for Amos II users to deal with. That is, Amos II must be enabled to extract retrieved ROOT data members in form of Amos II literal types. Moreover, they should be retrieved by queries.

Recall that AOD files contain several objects of type *TTree*. We will call them simply as *trees*. The trees are actually persistent (stored in a ROOT file) arrays of C++ objects of some class; we will refer to this class as *top level class of a tree*. If those classes were simple structures, probably they could be returned by ROW as vectors of atomic literal types (character strings, numbers, etc). But actually some of those classes are structures themselves, containing nested collection types.

To ease the management of data having complicated structure we decided to preserve that structure when representing objects as Amos II objects. Therefore while reading data from an AOD file ROW creates structures of corresponding topology. Moreover, functions are provided to navigate in these structures.

At the first stage of planning the architecture of the system, objects were intended to be mapped to Amos II *Vector*s, by representing an object aggregated into another object as nested vector structures. But we chose a slightly more complicated Amos II collection *Struct* because of several issues:

- To know the type of an object represented by a *Vector* requires to reserve one vector element for a type identifier, while *Struct* has a special field for the type identifier.

- Recursive printing to console of complex objects represented as vectors is inconvenient as the output may be huge and therefore hardly readable. By contrast a *Struct* is printed as an object address, rather than printing its (nested) contents. Printing circular vector structures furthermore causes indefinite looping.

5.2.2 Object represented as Amos II *Struct* type

The new collection type *Struct* type proved to be convenient. In Figure 5.2 you can see that *Struct* is similar to a vector containing attributes (elements) of type *Object*. Thus it can represent structures containing members of any type.

As it was mentioned in the previous section, we exploit conveniently attribute *typeo* (see figure 5.2) of type *Struct*. In this wrapper it is used to refer to the meta-data object (i. e., object containing names and types of its members, for details see section 5.2.4) which descibe the *Struct* object. This makes *Struct* objects self descriptive.

An alternative option would have been to represent each C++ object as an Amos II surrogate object and each new type as an Amos II type, i.e. to create new types for all of the classes encountered in the data file and create Amos II functions to represent data members. Objects of type *Struct* are basic and lightweight data holders more suitable for streaming because objects not satisfying the WHERE clause of a query are automatically disposed by the garbage collector. By contrast surrogate objects require much more additional space and processing and must be explicitly deleted when no longer needed. The sequence of actions using surrogates would be 1) read all data from the files and store it in Amos II database and 2) search the data using usual Amos II queries. By contrast, our approach is the following: 1) create *Struct* objects for each entry (see 3.3.2 for definition) 2) project the needed data fields for each entry and, if needed, store to permanent objects. As the number of objects stored in analysis data may be millions or billions, our approach proved to be faster, thus more suitable. Not least does it consume a minimal amount of memory as the automatic garbage collector immediately removes all *Struct* objects no longer needed.

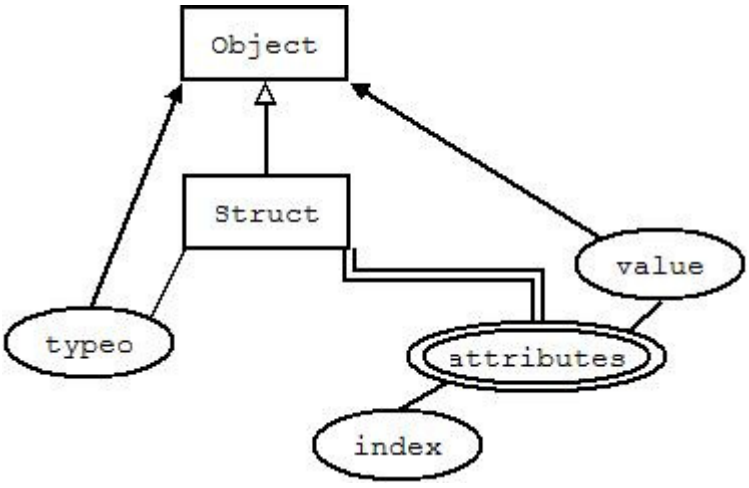


Figure 5.2 Layout of Amos II Struct object

5.2.3 Type mapping

Mappings to Amos II types are known at compile time for certain C/C++ types that are not mapped to Amos II *Struct*: primitive types, *string, char**, arrays, STL containers.

There is no need for every C++ class to have one-to-one mapping to an Amos II type. For example, there are many types in C for integer numbers (*short, integer, long*) while Amos II has only one built in type *Integer*. A subset of the literal types provided by Amos II is enough to represent any C++ class.

As Amos II provides type *Charstring* for storing character strings, it is natural to convert C type *char** and STL class *string* to type *Charstring*. (see Table 5.1). It is also of no use to represent differently an object of some class (e. g., class X) and a pointer data member (of type X*) pointing to an object of the same class because pointers are used as programming means and is not a database concept.

C++ types	Amos II type	meta-data type
`char*, char, string`	`CHARSTRING`	`ROOTAmosPrimitiveType`
`Int, short, long, unsigned int, unsigned short, unsigned long, unsigned char`	`INTEGER`	`ROOTAmosPrimitiveType`
`float, double`	`REAL`	`ROOTAmosPrimitiveType`
`Bool`	`BOOLEAN`	`ROOTAmosPrimitiveType`
arrays, STL containers	`VECTOR`	`ROOTVectorType`
classes, different from mentioned above	`STRUCT`	`ROOTStructType`
`X*`, where `X` is any type	type, to which `X` is mapped	meta-data type corresponding to `X`

Table 5.1: Mapping of types from C++ to Amos II

5.2.4 Stored meta-data

ROW is able to read objects of classes that were user-defined and not known at compile time. The C++ objects are represented as *Struct* objects. It utilizes ROOT file meta-data and ROOT library classes. The process of retrieving and storing this data is discussed in more detail in section 5.3.

As we may open several different ROOT files that contain objects of the same classes, or open the same file several times, re-using meta-data allows easy access to data from several different ROOT files. If different ROOT files containing the same kind of data had different wrapper meta-data, it would be hard to integrate data. For this purpose we store in the Amos II database ROOT/AOD the following meta-data information from several files (see Figure 5.3):

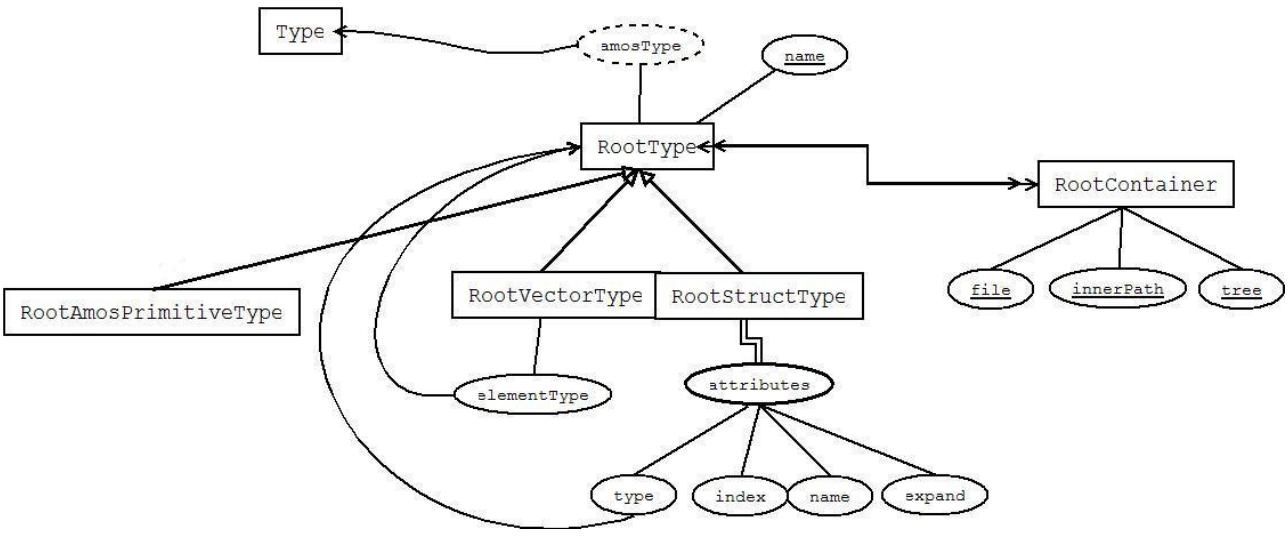


Figure 5.3 Metadata schema

The *ROOTContainer* represents a ROOT tree The tree is stored in a ROOT file named *File*. As a tree in a ROOT file may be stored inside some top level file directory structure, we need an attribute *InnerPath* to define the path to the tree inside the ROOT file. Thus a *RootContainer* is identified by the attributes *file*, *innerPath* (a directory path), and *tree* (a TTree object in that directory). The name *container* is chosen to foresee possibility of extending wrapper to deal with other kinds of ROOT object storages than *TTrees*.

- All C/C++ types (classes or a primitive types) that are used in a particular ROOT file are part of the file's meta-data. This information (i.e., names and members of a class) are extracted from the ROOT file before reading the actual data. The types are represented by the Amos II type *ROOTType*. *ROOTType* is an abstract type that has attributes to represent data common to all ROOT C++ types: *name* is type name and *amosType* is an object representing a built-in Amos II type. It refers to the type to which C++ type is meant to be mapped (see Table 5.1, column "Amos II type"). Different groups of types share specific features. Therefore meta-data of any type is an object of some of the following types (see Table 5.1 column "meta-data type"):
  1) *ROOTAmosPrimitiveType* represents meta-data of primitive C++ types. It contains no own attributes, but its *amosType* depends on the C++ type while the following *ROOTVectorType* is mapped constantly to *Vector* and *ROOTStructType* is mapped constantly to *Struct.*
  2) *ROOTVectorType* represents array-based types. It provides the meta-data of the single type of its elements. Formally, some STL containers (e.g. *map*) may have more than one element type in their definition. However, *map* was the only case we encountered among AOD data and it poses no problem as it is retrieved by ROOT as C++ template type *vector<pair>* where *pair* is name of a class that aggregates two classes: the one being mapped by a *map* object and the one being mapped to.
  3) *ROOTStructType* has four attributes describing each data member of a C++ class described by *ROOTStructType* object:
     1. *Type* refers to meta-data of a member thus allowing it be of any type
     2. *Index* is the position of the member in the data array within a *Struct* object
     3. *Name* is name of data member as it is declared in class described by *ROOTStructType*
     4. *Expand* is to enable breaking circular references. For explanation see section 5.3.2.

As the same type may be present in several different trees and trees may have several types, types *ROOTContainer* and *ROOTType* have many-to-many relationships.

5.3 Data retrieval process

Initially, for the sake of clearness, we will reference a ROOT tree by a more general term *container*.

5.3.1 Opening a container

To enable ROW to retrieve objects of types that were not known before (i. e. their meta-data are not stored in the Amos II database), the import of meta-data of ROOT classes must be performed prior to executing any queries upon the container.

The actions of meta-data importation are:

1. The stored procedure *openContainer* is called with parameters identifying the container externally.
2. If meta-data of this container is already present in the database (see "Container metadata in Amos II local database" in Figure 5.1), no meta-data retrieval is performed and the C++ module populates the local database with the inner meta-data in the container being opened.
3. Otherwise the stored procedure *openContainer* calls a foreign function that retrieves meta-data from the container and adds it to both the local database and wrapper's transient container meta-data storage that is defined independently of Amos II.
4. As retrieved meta-data may already be present in the local database, the stored procedure *removeDuplicates* is called that compares newly retrieved types' meta-data and already stored types' meta-data. If a newly retrieved type is equivalent to some type that was stored before, it is removed. Equivalence of types is defined by the following rules
   1) Primitive types are equivalent if they are mapped to the same Amos II type.
   2) Vector (collection) types are equivalent if they contain elements of equivalent type.
   3) Struct (class) types are equivalent if they have the same number of fields (attributes) and corresponding fields are equivalent.

5.3.2 Breaking circularities

A nontrivial task in wrapping ROOT/C++ objects is handling circular structures. Although all data could have been stored as surrogate database objects, this approach is cumbersome as we often need not store all the data from a file but some relatively small data selection. Type *Struct* has a disadvantage of not allowing circular structures, i.e., members in children structures cannot point back to parent structures. The restriction is mainly because of the reference counting based Amos II garbage collector [4]. Therefore, we have chosen to automatically track circularities while constructing mapped objects and issue error on noticing any of them. Moreover, to still be able to retrieve data, we decided to implement support for manual disabling of certain links (see chapter 6 about implementation) while preserving vital information.

5.3.3 Retrieving data

ROW data retrieval functions return *Struct* objects each of which contains data of a single object instance stored in a container. However, the user often requires only specific members (attributes) of the objects. To extract these attributes some Amos II data management functions are provided. The user is never required to call low level ROW functions directly. Instead the user issues an AmosQL query in terms of data management functions. After container has been opened such queries can be executed in the following way:

1. The user query is executed in terms of data management functions.
2. The data management functions either call the foreign function *row_get* to return a *Struct* object containing data of a single object stored in the container or *row_scan* that streams *Struct* objects containing data of all the objects in the container.
3. The wrapper constructs *Struct* objects using its internal meta-data (present in the wrapper RAM). This is more efficient than querying the Amos II database for meta-data.
4. Data management functions such as *getMember(Struct s, Charstring name) -> Object* use meta-data stored in the Amos II database to extract required members from the *Struct* objects returned by foreign functions.

   For example, consider a derived function definition:

   create function getEvents(Struct s) -> Struct as

   select getStructElement(getVectorMember(s, "m_pCont"), i) from Integer i;

Here s is a *Struct* is a meaningful variable if it represents an object of class *McEventCollection*. This class contains data member *m_pCont* which is a collection of objects of type *HepMC::GenEvent*. We extract it using the function *getVectorMember(Struct s, Charstring name)-> Vector*. We extract all elements of this collection by calling *getStructElement(Vector v, Integer i)-> Struct*. Thus the function returns all *HepMC::GenEvent* objects within a *McEventCollection* object represented by *Struct* s.

6 Implementation

6.1 Tools

The general tools used to work with AOD are the projects developed within CERN: ROOT, POOL and ATLAS Athena. We reviewed advantages and disadvantages of all these frameworks in sections 3.2 and 3.3. Our claims in those sections are supported by our personal tests and practical evaluation of the existing software.

As we implemented a wrapper for ROOT object trees, a compulsory tool which must be used in our work is, of course, ROOT. We based and tested our implementation on the newest currently stable versions, 5.08/b and 5.11. The other required software for us was Amos II DBMS. As ROOT is a framework for C++ and Amos II has a full support for call-in and call-out from C, we used C++ language to provide the bridge between the two systems. Some of the functions were implemented using AmosQL.

Moreover, we aimed to develop a platform independent wrapper version. The two platforms considered were Microsoft Windows and Linux. We used GNU C++ Compiler (gcc) version 3.3.5 and Microsoft Visual C++ 6 to compile the wrapper for corresponding platforms. Despite of several minor issues regarding ROOT and Microsoft Visual C++ 6 compiler compatibility, we succeeded to develop a common source compiling on both platforms.

6.2 Query processing

Consider as an example two possible scenarios of requests from Amos II users. The first scenario is opening an object tree in a ROOT database. More precisely, a user decides to work with data stored in file “MyROOTFile.ROOT” which has an internal directory named “DirectoryInsideROOT” and a tree, or container, named “MyContainer”. In ROW, this container must first be opened and type information stored in the transient container meta-data storage. After the container is opened the user may decide to retrieve the first entry in this container.

To open the container, the user needs to call the procedure *openContainer*:

```
AODWrap 1> set :myROOTContainer = openContainer("MyROOTFile.ROOT", "DirectoryInsideROOT",
"MyContainer");
10 types imported;
```

The parameters of *openContainer* describe the location of a container in ROOT file hierarchy.

Consider what is done in the function *openContainer()*. First, we want to know if this container exists at all as a tree in the given file and the given directory. Thus we must call ROOT to actually open that container. We use a foreign procedure *row_openContainer* for that reason.

The external opening routine checks the existence of a container and, if it exists, returns a success value. The foreign procedure has arguments consisting of the file path in the OS, path inside ROOT file.

Once a container is opened, calling any of the lower layer routines with the same key will not require a repeated opening of the container, as it will be stored in transient container metadata storage.

After the container is opened reading is not possible yet as there is no representation of ROOT types in the database as meta-data objects. Thus, implicitly after opening, the AmosQL layer

wrapper part calls the C++ interface again to retrieve the types encountered in the container that was just opened. That is, it retrieves the types of top-level objects stored in that tree as well as types of all the types of its member objects, etc. The AmosQL layer also performs a type matching against the types already defined before in order to avoid duplication. It finishes opening of the container by removing the types which are duplicates of some previously defined ones, thus creating a new *ROOTContainer* object, setting the necessary values of stored functions in the database, and returning the container object for a user.

Now let's watch the event flow when a data retrieval query is issued. Let's say that :*myROOTContainer* was successfully opened and the user issues the query:

```
select get(:myROOTContainer, 0);
[struct #999];
```

In this case the processing proceeds as follows:

1. Amos II calls function `get` which is simply a derived function defined in upper wrapper layer.
2. The function substitutes *:myROOTContainer* with a key that is associated to this container and calls a foreign function named *row_get*:

```
row_get(<"MyROOTFile.ROOT", "DirectoryInsideROOT", "MyContainer">, entryNumber);
```

3. The Amos II optimizer takes action for the multi-directional function *row_get* and decides that it is more efficient to get entry number `entryNumber` of the container by calling foreign C++ subroutine `row_get` (bbf binding) or to call the subroutine `row_scan` which returns all of the entries (bff binding) and then filter out the entry with a particular key. Say, it decides to use bbf binding. Then a foreign subroutine `row_get()` is called with the same arguments.
4. The function for getting an entry is executed in the lower (C++) layer. A ROOT library function that retrieves all the data and constructs a binary C++ object of a particular structure is called.
5. The lower layer part uses the information which was collected upon opening the file to navigate inside the created object and creates an Amos *Struct* representing it.
6. A handle of a newly created *Struct* is emitted to upper layer.

6.3 C++ layer

In this section we describe the lower layer of the wrapper (the C++ module) implementation in detail.

6.3.1 C++ layer organisation

The lower layer wrapper part provides the only necessary functionality to access ROOT via its C++ interface, to navigate in ROOT files, and to retrieve data stored in them. Since our aim was to wrap only ROOT object trees we used (not very strictly) object oriented approach to implement this part.

The classes used in the implementation roughly correspond to those described in the meta-data schema in section 5.2. Although the call-in interface of Amos II enables querying for any data

stored in the database image, for efficiency ROW additionally maintains an internal transient container meta-data (which actually duplicates part of the meta-data stored in local Amos II database) together with the handles to meta-data objects stored in the image. It was done mainly because of efficiency reasons as the retrieval of data members in an object should be fast. Also certain properties which are needed for the lower layer make no sense for the upper layer and the user, for example, offset of certain member in a class. In other words we manipulate persistent meta-data storage in upper layer and it's transient equivalent in lower layer.

The routines implemented in the lower layer are grouped into the following parts:

1. ROOT database navigation routines. The lower layer provides navigation within ROOT files which is only possible by calling ROOT functions from C++. The routines are scanning for all the containers stored in a ROOT database (file), container opening, and container closing.
2. Routines providing synchronization between upper and lower layers. Since we have representation of meta-data information in both upper and lower layers, we must ensure that they are always synchronized. This type of methods include exporting (and creation) of class information of the objects in a specific container to the upper layer, updating type information (in case upper layer decides to remove a certain type object), setting the database connection handle (to be able to call Amos II functions), etc.
3. Class meta-data retrieval routines. The most important is a method that uses ROOT streamer information to automatically construct class information for every object stored in a file. See the following subsection.
4. Data retrieval routines. For example, a routine to create a *Struct* object from a given binary C++ object retrieved by ROOT. The mapping between objects of any of the C++ classes which are stored in ROOT classes and Amos II type *Struct* are explained in Section 5.2. The functioning of the *Struct* creation routine is further explained in the next subsection.

6.3.2 Data reconstruction using ROOT

Having opened a branch in a *TTree*, one needs to call a method, *TBranch.GetEntry()* to recreate in the RAM the object stored in that branch and get access to it through a pointer. Yet, to be able to interpret data recreated under the root pointer one must know the inner structure of it. The pointer structure is automatically created by ROOT's streamer when opening the file where the tree is stored. In the RAM ROOT maintains a list of classes (*TClass*), both those used internally and those loaded from a ROOT file. This list can be accessed by searching for a class with a specific name (method `TROOT::GetClass(const char *)`). Member names and other properties can be further browsed using the ROOT class methods *TClass* and *TStreamerInfo*. Special classes are provided for representing collections (*TVirtualCollectionProxy* and the derived classes). These classes enable description of a collection of practically any type of collections that could occur and most importantly, the C++ STL collections (vectors, sets, maps, ...).

ROW calls the ROOT functions only once, on importing the meta-data of the types in a container. After the meta-data is imported, the wrapper uses its own efficient transient container meta-data storage.

Navigating through binary representation of retrieved transient C++ objects is done using raw addresses and offsets of members. As the variety of possible classes is so big in C++ neither ROOT aims to be able to stream all of them (though it covers a large variety of classes) nor do we claim to handle all the types handled by ROOT. However our wrapper does provide enough flexibility to extract the following members of any class:

1. Primitive members.

2. Arrays of any type.
3. Aggregated objects.
4. Base class members in objects of an inherited class.
5. Pointer members.
6. Elements, if the class is as an array-like class.

Note that array-like classes (STL containers and arrays) are treated more conveniently than general ROOT C++ classes as ROOT recognizes them as collections. It means that data is extracted from array-like objects independently of their implementation and used needs not to know their internal structure.

The basic algorithm to build a Struct object applies the ideas described in chapter 5 to construct structures representing an object retrieved from a container. It is a modification of depth-first search (DFS):

- If a data member of an object is a primitive data member, it is converted to an Amos II object explicitly according to the built-in type mapping table. The algorithm removes pointer information, e.g., both data of type *int* and *int** are represented as type *Integer* in Amos II.
- If the member is an object or a pointer to an object, the algorithm recursively proceeds to create *Struct* objects from the bytes located at the member's address and the *Struct* is inserted into the parent object.
- If the member is some collection, for example, an STL vector, data is retrieved using methods *PushProxy()* and *Get()* of class *TVirtualProxy*, which handles all types of collections. Inner objects are then converted one by one recursively using the same algorithm.

An interesting issue is how to handle circular references of pointers. Two problems are solved:

- In order to avoid infinite loops a cache of already accessed objects is maintained during structure construction. The cache is represented as a hash map with a key consisting of the class of a C++ object, a pointer to the object, and a pointer to an already created Amos II structure storing the converted structure. Whenever an object that was previously created is encountered in the cache a pointer to the structure which was previously assigned for it is returned.
- Because the Amos II garbage collector is not able to reclaim object structures with circular references, the object creation routine must also track if the object creation path does forms a cycle. It is done without losing almost any speed at all by flagging all the objects on the current search path and unflagging whenever the path object is backed up (left). In case of a cycle, when a flag is encountered that was placed on the path, the structure that is being created is deleted, an error message issued, and the user is able to manually declare disabled fields causing circularities. This is done by setting that field's expansion attribute to 0 – see meta-data schema in section 5.1.. Otherwise, the structure is successfully constructed and is emitted to Amos II connection as a function result.

6.3.3 Calling Amos II back from the wrapper

Amos II call-in and storage interfaces are extensively used all throughout the lower layer implementation. The lower layer is responsible for creating instances of the new surrogate Amos II type *RootType* and the literal types *Struct, Vector, Integer, Charstring,* and *Real* which are all Amos

II types. Furthermore, it sets function values for *RootType* and accesses the Amos II database image to ensure synchronization between the two layers. The basic data structures used are Amos II *Struct*s and *Vectors* which are the transferring medium between the lower and higher layers as well as between the wrapper and Amos II environment.

Below we give the list of the most important Amos II call-in interface C functions called back from ROW and short comments:

`make_struct(size, type)` – creates a transient *Struct* object with *size* members and meta description refered by *type*. This object can represent most of the C++ objects stored with ROOT;

`new_array(size)` – creates a transient array (*Vector*) object of the specified size. Arrays were used to represent any C++ collections wrapper was implemented for;

`struct_set(bindtype, index, object)` – sets the *index*'th element of a *Struct* object. It is used to create referenced objects. A typical example of using this function is a recursive calls from the object creation method *fillObject.* Its return value is a pointer to another *Struct* or a *Vector*: `struct_set(varstacktop, i, fillObject(...))`;

`a_seta(array, index, object)` – sets the index'th element of an array. It was used to fill arrays;

`struct_free(s)` – release reference to the Struct s;

`a_free(object)` – release reference to the specified object;

`a_setf(source, destination)` – makes a reference to destination from source. This is similar to assignments but the system also updates the object reference counters.

`a_emit(context, tuple)` – emits a result placed in a tuple as the result of a foreign Amos II function;

`a_newinteger(),a_newdouble(),a_newstring()` - create and initialize literal objects;

`a_gettype(connection, name, stopOnError)` – retrieve the specified Amos II type identifier.

6.3.4 Caching

To speed up data retrieval in the cases when a single entry can be accessed many times in the same query, a simple caching mechanism is implemented. A fixed number of the most recently accessed entries can be remembered. This gives a dramatic improvement on non-trivial queries when Amos II must read each entry several times.

6.4 The AmosQL layer

This layer is physically organised into five files:

- *external.amosql* contains the foreign function definitions to access the lower layer,
- *schema.amosql* contains the meta-data schema definitions,
- *container.amosql* contains *ROOTContainer* type management functions,
- *struct.amosql* defines the *Struct* literal type and
- *aoddefs.amosql* the main script that includes all previous scripts for convenient script execution.

The upper layer code is small compared with the lower layer. The upper layer interacts with C++ layer using it's foreign function interface. A foreign function definition is realized using the following calls [3]:

a) Define the implementation in C of a foreign AmosQL function, e. g.,

```
void row_open(a_callcontext, a_tuple)
```

row_open opens a container to be used in AmosQL queries;

b) Register the foreign function name to Amos II:

```
a_extfunction("row_open", row_open);
```

c) Define the foreign function signature as AmosQL:

```
create function row_open(Charstring file, Charstring innerPath, Charstring tree)->Integer
as foreign "row_open";
```

`row_open` is one of the typical lower layer's interface functions. Although they could be called directly by the user, the upper layer provides a more convenient encapsulation of the lower layer's interface as well as additional functionality. It manages the *ROOTContainer* Amos II type defined in the meta-data schema. Instead of always repeating a compound key identificator (file, innerPath, tree) the user can instead operate with a single *ROOTContainer* object. *ROOTContainer* functions allow to access both key and type information associated with a particular container.

The duplicate type removal stored procedure is also implemented in upper layer.

The upper layer uses a trigger for setting which members of a certain type should not be retrieved. This provides an escape from forming of cyclic objects of type *Struct* which are not supported by Amos II garbage collector. As meta-data objects are persistent objects in the database, the lower layer implementation maintains a transient copy of meta-data in the memory. Whenever the object in the image is changed, lower layer must be informed to also update its information. It easily done by overriding default Amos II setter using function *set_setfunction()* with manually programmed code that performs a suitable call to external layer (see Amos II documentation, [3, 4, 17]).

6.5 Limitations

In this section we discuss some system limitations.

6.5.1 Reference breaking

Consider two types, A and B:

```
class B;

class A {
      B * b;
```

```
};

class B {
      A * a;
};
```

In the case when some instances of some type contain circular references and for other instances of the same type there is no such circularity some information can be lost.

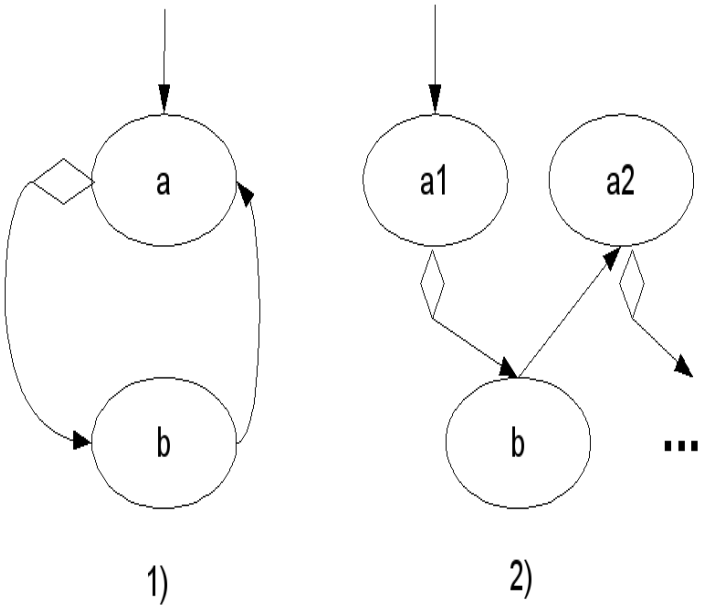


Figure 6.1. Circular references

In figure 6.1 an instance of a structure with circular references is displayed. *a*, *a1* and *a2* represent instances of type *A* (containing attribute of type *B*) and *b* represent instances of type *B* (containing pointer to instance of type *A*). Assume that instances a and a1 are top layer objects of respective structures 1) and 2). Presence of such instances in a file read by ROW would require the user to break manually either references from *A* to *B*. This makes access to attribute b to be lost. Alternatively, the link from *B* to *A* could be broken, which would cause loss of access to instance *a2* (assuming that it is not referenced by anything else).

We offer a simple solution[5] in special cases when objects of type A have a field that uniquely identifies them. For such cases instead of completely breaking a link (i.e. storing a NIL for each member a of B) we leave ghost objects at b.a. They are of the same form as object a, but have only

[5] In this case a user needs to issue the following commands:
set expandable("B", "a") = 0;
set expandable2("A", "id") = 1;
Here "id" is a field uniquely identifying objects of type A. A more detail description of usage of the attributes and functions is given in the ROW definition files.

their identifier field not NIL. Restoring a whole object from such a "ghost" object is, on the other hand, also left for a user.

6.5.2 Limited variety of accessible ROOT trees

Another restriction is on variations of the trees handled by ROW. Currently it only handles those object trees that have a single top branch. This was done to simplify navigation in the ROOT files. Otherwise a key specifying an object would additionally require one or two strings describing the object's position in a tree. For the AOD files where we performed our tests our implementation was satisfactory. Browsing in trees can be extended in future by making minor modifications in our code.

6.5.3 Bulky retrieval of data

As it was explained in Section 3.3.3, collections of objects in ROOT trees are stored in a branch. If an object has a member which is also an object, a sub-branch can be created to hold this member for each of the "upper" objects. Currently it is only possible to retrieve a whole top level object in a branch and this object can be big. A *Struct* representing it is constructed and passed to the user. Afterwards the user is able to use the *getMember(structure, memberName)* function to access the relevant data. If the user's members to be accessed are known in advance, only the desired substructure could be constructed instead of first building a whole top object structure and then selecting particular substructure in it. To put it simply, the current work of the wrapper is as follows: first a tree (a *Struct* object) with a possibly huge number of branches is returned, then certain sub-trees are picked from that tree. If the sub-trees are considerably smaller than the tree, it is more efficient to retrieve (at C++ layer of the wrapper) only the relevant sub-trees and not to generate the branches of the rest of the top tree. This extension could make a significant improvement in reading time if the split level (defined in 3.3.3) of ROOT branches is high. However it would require analysis of how it can be implemented with the Amos II query optimizer. For example, the optimizer could decide that only a sub-sub-sub-objects are enough to be retrieved instead of fetching the whole objects and then discarding most of the data.

Additionally, as our experiments show (see chapter 7) the major part of time is taken by construction of *Struct* objects. The fewer objects were constructed, the better performance could be expected. So even if branches were not split and ROOT could only be able to read a whole top-level object at once we could have an improvement.

6.5.4 Memory deallocation problems in ROOT

Until version 5.13 ROOT does not support automatical memory deallocation for emulated objects. This is a serious limitation for our wrapper but also for ROOT. As soon as it is implemented, almost no changes need to be done in the wrapper code to call the deallocator.

7 Application to Athena AOD files

Although LHC and ATLAS detector are not running yet, software to simulate AOD data exists. One of the most widely used packages is ATLAS Fast Simulation Package (AtlFast, [10]) which is based on Athena Framework. It produces event datasets using Monte Carlo reconstruction chain. It should be pointed out that there is no strict common layout for AOD files. To store particle information AtlFast uses a number of containers (*ElectronContainer*, *PhotonContainer*, etc.) which are internally represented as ROOT trees and using *AthenaPool*. Another large set of simulated files was generated in Rome workshop in 2005 [15] one generator of which was called DC2. We also used some of AOD datasets simulated using the same software on NorduGrid. This type of files has relatively few containers inside, a single *McEventCollection* contains all the event data. It is possible to browse the file structure using ROOT's graphical browser (figure 7.1).

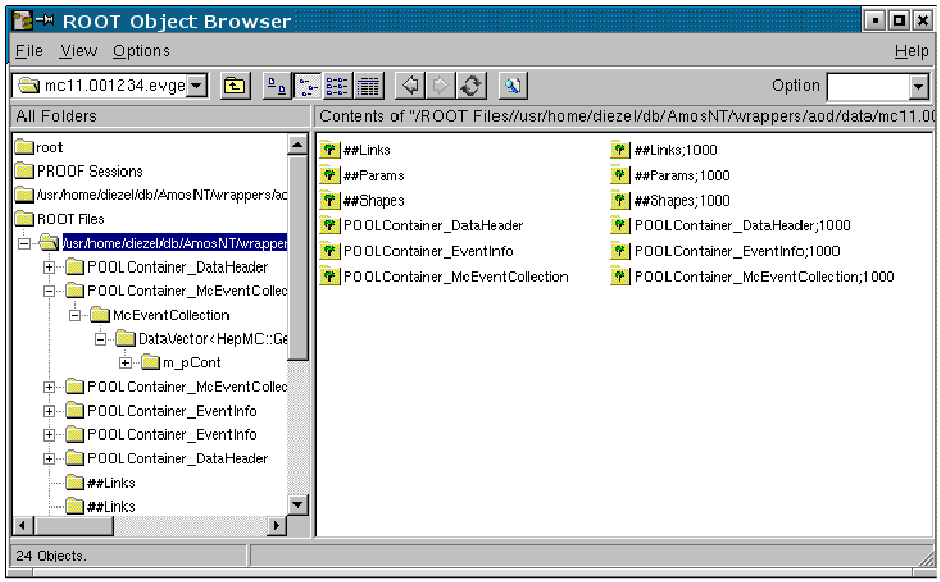


Figure 7.1: Snapshot of ROOT browser window

In all the datasets we worked with the names of object containers begin with prefix "POOLContainer_". *McEventCollection* is a class developed by HEP (High Energy Physics) software [9]. There is a ROOT tree with a single object branch created to store it. In figure 7.2 you can see a structure of a single *McEventCollection* object. The files we tested our wrapper on contained 1000 or 10000 of records in *McEventCollection* branch.

The other two meaningful containers in this type of files store *EventInfo* and *DataHeader* objects. From *EventInfo* meta-data describing each of the events can be extracted and *DataHeader* holds class structure information.

Of course, in order to access and use the data, the user should be aware of the layout (schema) of the file. The physical layout can be investigated using ROWs navigation interface. However, the semantics of containers need a higher level interpretation. Much information is also provided by class member names, which are actually private data members in C++, for example *m_alphaQED* in class *HepMC::GenEvent*. It is important to note that the wrapper does not allow calling methods of objects. For example, *McEventCollection* contains a method *sort()* which is not callable from ROW. Thus only the data members contained in an aboject can be retrieved and tthis is a drawback comparing to accessing objects through Athena having access to all possible methods.

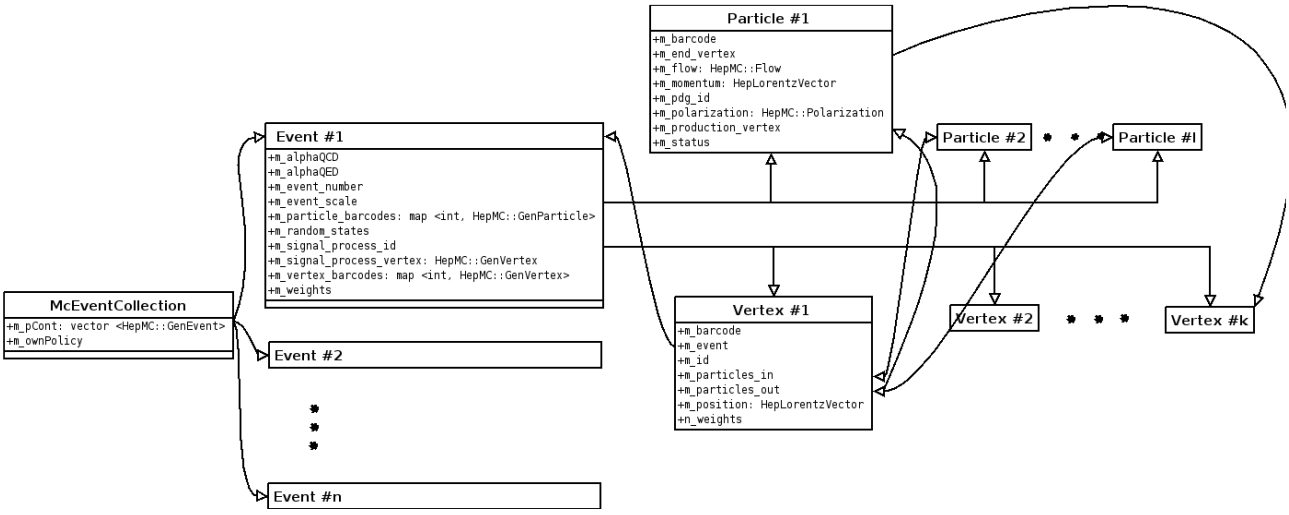


Figure 7.2: Structure of a single record for object McEventCollection

7.1 How to query AOD files

Two examples were presented in section 2, which also used AOD files. The examples skipped many details which should be clearer now. Let's focus on retrieving of some data from a Rome AOD file. Our example file is named "`rome.004100.FAST_AOD.T1_McAtNLO_top._00001.pool.ROOT`" and was downloaded from the ATLAS AOD sample site [15]. To open one of the containers, *McEventCollection*, we issue the following queries to Amos II:

```
AODWrap 1> declare ROOTContainer :c;
<#[OID 905 "ROOTCONTAINER"],AMOS_C>
0.000168 s
AODWrap 1> set :c = openContainer("rome.004100.FAST_AOD.T1_McAtNLO_top._00001.pool.ROOT",
"", "POOLContainer_McEventCollection");
<Warnings by ROOT>
20 types imported
0.553148 s
```

Reading an entry with an *McEventCollection* is then as simple as

```
AODWrap 2> declare Struct :s;
<#[OID 985 "STRUCT"],AMOS_S>
<#[OID 905 "ROOTCONTAINER"],AMOS_C>
7.4e-05 s
AODWrap 2> set :s = get(:c, 10);
```

The output is however:

```
LOOP DETECTED! ROOT OBJECT WRAPPER CANNOT HANDLE LOOPS! Please use set
expandable(ROOTTypeObject, memberName) = 0 to disable members causing loops.
```

```
The detected loop is: McEventCollection->(Start of cycle)m_pCont[0]-
>m_particle_barcodes[0]->second->m_end_vertex->m_event-> (back to ...)m_pCont[0](End of
cycle)
0.111242 s
```

After inspecting the structure of object, the user realizes that loops are caused by a back references to parents and relationship between *GenVertex* and *GenParticle*. To be able to continue reading one must break the loops:

```
set expandable(getType("HepMC::GenVertex"), "m_event") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_in") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_out") = 0;
set expandable(getType("HepMC::Flow"), "m_particle_owner") = 0;
```

And now the `get()` routine can be called successfully.

The default calls for getting type information are provided by functions *struct_type()* and objects of superclass *ROOTType*. We can simplify access of particular members or sub-members of an object. In a couple lines in AmosQL we defined the following user function for navigating *Struct* objects (see Appendix A, test C for the declaration):

```
particle(Struct s, Integer i)->Struct
```

This function returns particles at a certain position of the first event in *McEventCollection* assuming that *Struct s* represents an *McEventCollection* object. In the similar manner, any of the member functions can be defined to make them considerably shortened.

Finally we can check what information is stored for the event:

```
AODWrap 6> set :c = openContainer(”rome.004100.FAST_AOD.T1_McAtNLO_top._00001.pool.ROOT”,
“”, “POOLContainer_EventInfo”);
6 types imported
0.129168 s
AODWrap 7> set :s = get(:c, 10);
0.121476 s
AODWrap 9> toVector(cast (getmember(:s, “m_event_ID”) as Struct));
{{”m_event_number”,10},{”m_run_number”,0},{”m_time_stamp”,0}}
```

7.2 Performance evaluation

To evaluate the performance of ROW we selected a number of containers from AOD files of different sizes and ran various data retrieval functions on different types of files. We performed 3 tests of increasing complexity. Tests A and B execute function *get()* to fetch random entries from a single or several containers. Test C shows how efficiently ROW selects specific data members. Queries of the tests are shown in Appendix A.

The tests were:

A - *McEventCollection* container in file named `dc2.003007.evgen.A1_z_ee._00092.pool.root` generated on ATLAS data challenges phase 1 at NorduGrid. It has a very complex structure with

hundreds of particles and vertices in each event. The test function was retrieving 1000 random (uniformly) records.

B – *ElectronContainer* containers in 118 AOD files from a Rome data set. The test function was to retrieve 5000 random entries from all the containers, first uniformly by randomly choosing one of the containers then getting one of its entries. This test was presented in Chapter 2.

C – The same conditions and data as in the test C, only additional data extraction is done in Amos II afterwards. That is, for each of the 1000 records the momentum of each particle in a record (event) was extracted and printed from a file `dc2.003007.evgen.A1_z_ee._00092.pool.root`. The momentum consists of only 3 doubles, dx, dy and dz. Each record had set of approx. 875 particles in average. Thus approximately 15 MB of data was actually extracted. The query was written without any optimization accessing *Struct* members by names.

There are two lines for each test. The first line of values represent results when caching mechanism is not used and the second one shows how caching speeds up the performance when 1000 last entries are kept in the cache.

Note also that (Appendix A) to get optimal results we use precompiled queries (i.e. queries defined as derived functions in AmosQL) as well as random numbers generated in advance so that query optimization time is not included in the tests.

	Caching	Accessed records (overall records)	Overall objects (structs and vectors) created	Size of accessed records in ROOT, uncompressed	Time[6] of ROOT GetEntry(), s (% of all)	Time to create objects, s (% of all)	Query execution time, s (% of all)
Test A	uncached	*1000(10000)*	7680052	138423 KB	74,436 (67%)	108,275 (98%)	110,418 (100%)
	cached	*1000(10000)*	7371604	132864 KB	66,199 (68%)	95,714 (99%)	96,138 (100%)
Test B	uncached	*5000(5336)*	834318	60251 KB	7,186 (64%)	10,233 (91%)	11,213 (100%)
	cached	*5000(5336)*	552172	39877 KB	4,774 (66%)	6,848 (96%)	7,138 (100%)
Test C	uncached	*1000(10000)*	7675144 (~875000 structs projected)	138334 KB (~15380 KB projected)	44,669 (37%)	79,114 (65%)	122,024 (100%)
	cached	*1000(10000)*	7366666 (~840000 structs projected)	132775 KB (~14760 KB projected)	42,330 (37%)	73,448 (64%)	115,497 (100%)

Table 7.1, Performance evaluation

In test A we see that wrapper reconstruction part takes roughly the same time as ROOT object reading. A more detail graph of performance time is given in Figure 7.3. Taking into account that we can not exploit such properties as direct addressing while constructing Amos II vectors, we consider the wrapper performance very satisfactory.

6 Time was calculated in seconds using default Linux function gettimeofday() and default Amos II output on idle Linux. Note, that times in the rows are accumulated: object creation includes also reading with ROOT and query includes all operations .The overall time includes both query optimization and query execution (when the result is constructed) time. The time to open/close containers or to generate random values is not included. The evaluated queries are denoted **in bold** in Appendix A.

Test B measured performance of the wrapper with entirely different conditions: we had many relatively small input files and we scanned them all at once. The main difference from the case A is opening time. We didn't include file opening (and type importing) time in the table. For test A (570 MB) it takes approximately 0.55 s, for test B – 22.51 s (118 files, 774 MB overall, but *ElectronContainers* contribute only around 70 MB to this amount).

Finally, test C evaluated how fast Amos II queries can be executed on extracted data structures. While almost all values were very similar to those of test A, extraction of certain fields in Amos II took a considerably longer time, namely, ten times longer than work of wrapper. We believe that it is possible to achieve much better performance in Amos II by either optimizing a query itself or implementing a specific handling of *Struct* for the Amos II optimizer. Unfortunately this task is also out of the scope of our project. Also see the other possible optimization in section 8.2.

The three tests which evaluated ROW included only one complex query, namely, the last one. More advanced queries using conditions and involving many containers can be easily constructed as well. In this case, however, an attention should be paid to correct construction of a query. Otherwise, the slow *get()* function can be called many times with different arguments. We partially resolve this issue by implementing a simple caching mechanism. In the example queries caching saves up to 36% time. We found that caching is extremely effective for even more complex queries when Amos II optimizer cannot find a strategy to read each entry only once. The speedup is up to 500%, however, both due to lack of time and for the reason that our aim is to test only low level functioning we do not present that kind of experiments.

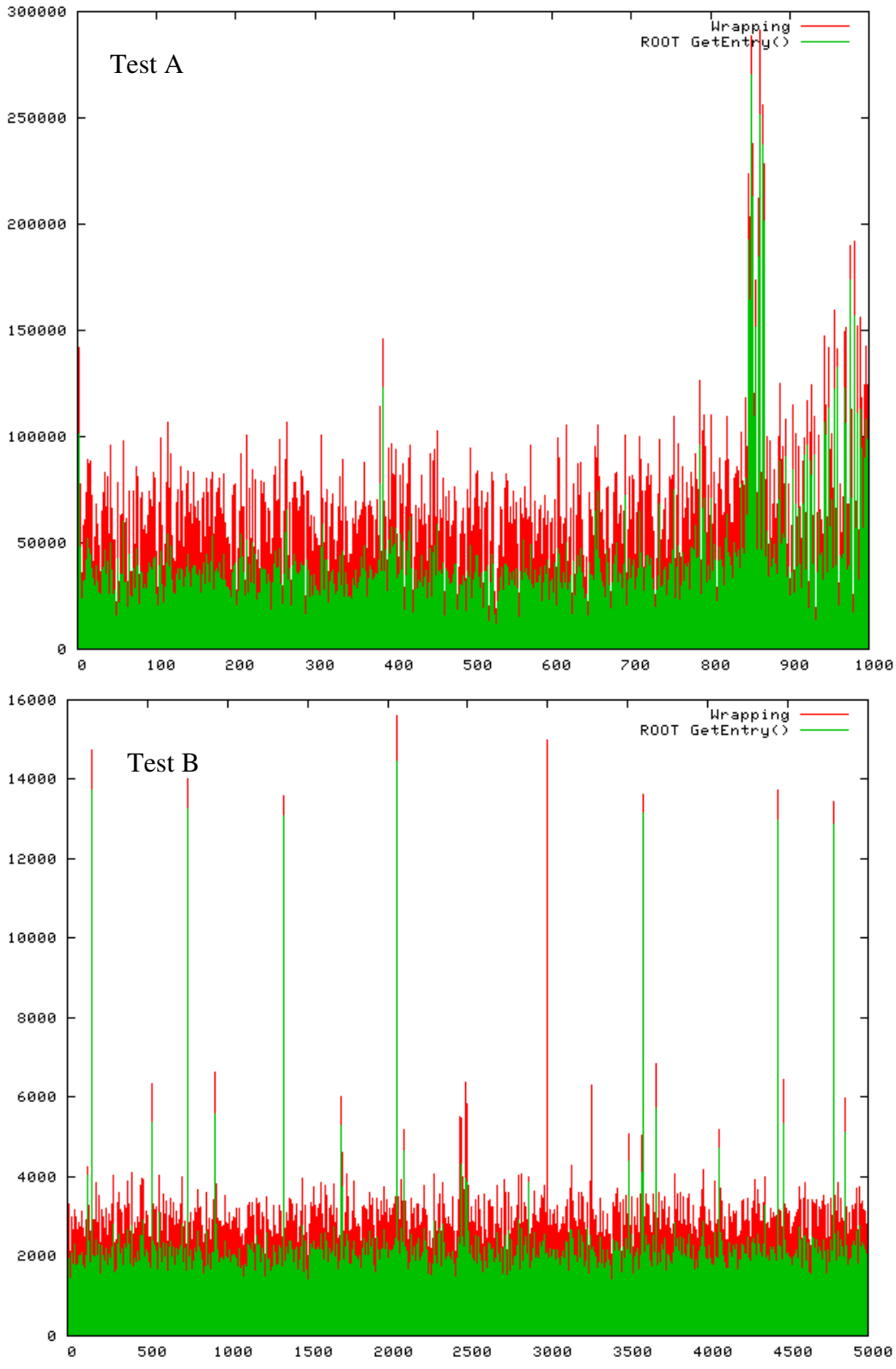


Figure 7.3, Time to perform a query in tests A and B. Horizontal axis - number of get() call, vertical - time in microseconds. GetEntry() time (lines below) corresponds to time of internal ROOT function. Wrapping time is time of GetEntry() plus time to create an Amos object from a record. The peaks probably indicate disk accesses.

8 Conclusions

In this project we have implemented a fully functioning wrapper for a broad extent of analysis data files that are created with ROOT. This wrapper in turn broadens the capabilities of object-oriented functional relational DBMS Amos II. It enables physicists or programmers to combine the flexibility of ROOT storage manager and speed of DBMS queries.

During our work we investigated the best ways to work with data sets used in particle physics, particularly in the data generated by LHC without using Athena framework software. We found that the simplest and the fastest way is to base data retrieval on ROOT.

Furthermore we managed to map the structures and metadata used in ROOT, i.e. the C++ objects organised into containers, into entities and data structures in Amos II.

Lastly we performed tests of our wrapper on currently existing simulated data sets of ATLAS experiment and found that our implementation runs fast and is general enough to open and read data generated and tested with Europe's leading physics computing software to date.

We hope that our efforts and insights were useful in further ROOT, Amos II and other related software development.

It seems that there is much to be improved in both LHC software and Amos II. Firstly, projects POOL and Athena should improve and realize their own goals, namely become suitable for many users to work within or outside Grid environment. As we have seen during our project, only ROOT provides both satisfactory functionality and documentation.

As for the Amos II part, much of the improvement could be obtained optimizing ROW queries passed to. We have shown in our experiments that queries formulated in naïve way performs rather inefficiently accessing many *Struct* members, as without caching the ROOT files are accessed many times. A possible improvement from the wrapper part would be to return only certain subset of data instead of a whole record. In this way we would not save any time in ROOT part (3 row from the bottom, Table 5.1), but Amos II would then receive *Structs* of smaller size to perform queries on and consequently would spend less time.

One particular drawback of reading ATLAS generator data files using only ROOT is that while we are able to retrieve data members, we loose access to methods operating on them which are all implemented in the Athena framework. It means that the necessary functions have to be redefined if one wants to perform analysis on data wrapped for Amos II. However, many projects in CERN are using separate data and software to process it. It means that in the future not only data members, but also all functions can be accessed with more kinds of software. This functionality is to be provided in Reflex package started by project SEAL [21] and now continued within ROOT and efforts of POOL and Athena framework developers. In this case it could be possible to further extend our ROOT object tree wrapper to also automatically wrap object methods.

On the other hand hierarchy and functionality related with cataloguing, uniquely indexing objects, etc., cannot be correctly supported using only the lower layer, ROOT, instead of Athena or POOL. It means that so far Amos II user has to maintain management of his own set of physical datasets. As soon as POOL becomes more flexible and easier to use, two kinds of improvement can be made:

A hierarchy navigation wrapped with our software creating an additional module and/or functions in the schema.

- An alternative wrapper can be implemented using only POOL or Athena interfaces.

Lastly we leave recursive links problem only partially resolved. That is we only detect loops in data and leave for a user to manually break them. There are also two ways to make this issue easier:

- Change the Struct data type in Amos II to support cycles. It would require extending the garbage collector.

As any programs, our wrapper is not likely to be a bug-free. As it was noticed in Section 6.5.4 a very serious limitation is that ROOT's ability to free the allocated memory does not work yet. There are some compatibility issues for reading ANSI strings in early versions of C++. For the ROOT version we implemented (5.11) we have not noticed any other bugs. However, ROOT is an evolving system, sometimes not supporting even its older versions, so to keep up with state-of-the-art wrapper should be continuously improved.

Finally we are looking forward to applications of our wrapper. It provides only intermediate interface which we hope is easy to cope with for a programmer who knows Amos II but perhaps not for a physicist. Future work should investigate the best ways to apply our work for particle physics analysis, provide a good test background for both ROOT and Amos II and possibly offer improvements for the wrapper. We expect that integrating and using our software in the Grid will also play a role in NorduGrid development and open new areas of work at UDBL in Uppsala University.

9 Acknowledgements

We would like to thank the two ROOT developers, Philippe Canal and Axel Naumann, working at CERN who never hesitated to answer and advice on the principles of object streaming and programming subtleties in ROOT.

Also we received a great help from professor Tore Risch who made many useful insights when carrying out the project and writing the report as well as from our supervisor Ruslan Fomkin who introduced us to principles and development of physics software.

10 References

1. T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.
2. R.Fomkin and T.Risch: Managing Long Running Queries in Grid Environment, 1st Intl. Workshop on GRID Computing and its Applications to Data Analysis (GADA'04), Lacarna, Cyprus, Oct. 2004, in R. Meersman et al. (Eds.): OTM Workshops 2004, LNCS 3292, pp. 99–110, 2004
3. T.Risch. Amos II External Interfaces, Uppsala University, 2000.
4. T.Risch. ALisp User's Guide, Uppsala University, 2000.
5. J. Tysklind: Wrapping a Scientific Data Management System Uppsala Master's Theses in Computing Science 301, ISSN 1100-1836, 2005.
6. ROOT Users Guide v5.08. CERN. Online version: http://ROOT.cern.ch/ROOT/doc/ROOTDoc.html.
7. F. Rademakers, R. Brun. ROOT: an Object Oriented Data Analysis Framework. Linux Journal. 1998.
8. J. Knobloch. LHC Computing Grid. Technical Design Report. 2005. http://lcg.web.cern.ch/LCG/tdr/
9. M. Dobbs, J.B. Hansen. HepMC: a C++ Event Record for Monte Carlo Generators. Comput. Phys. Commun. 134 (2001) 41.
10. E. Richter-Was, D. Froidevaux, L. Poggioli, ATLFAST 2.0 : a fast simulation package for ATLAS, ATL-PHYS-98-138, November 1998. (http://www.hep.ucl.ac.uk/atlas/atlfast/)
11. See Wikipedia, ATLAS experiment, http://en.wikipedia.org/wiki/ATLAS_experiment (as of Jun. 12, 2006).
12. See LHC homepage, http://lhc.web.cern.ch/lhc/.
13. Atlas Athena Developer's Guide. Draft. Online version: http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/
14. See ATLAS wiki pages. https://uimon.cern.ch/twiki/bin/view//EventDataModel.
15. ATLAS AOD simulation sample site, http://www.nikhef.nl/pub/experiments/atlaswiki/index.php/AtlasATLASTopPhysicsSamples .
16. See POQSEC project homepage. http://user.it.uu.se/~udbl/poqsec.html.
17. See Amos II homepage at UDBL. http://user.it.uu.se/~udbl/amos/.
18. See POOL homepage. http://pool.cern.ch (as of Aug. 9, 2006)
19. Amos II User's Manual. http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html.
20. The Grid Café - What is Grid?. CERN. (As of Aug. 26, 2006).
21. See SEAL homepage. http://seal.cern.ch (as of Aug. 26, 2006).
22. ATLAS Data Challenges homepage. http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/DC/index.html
23. Athena framework. https://twiki.cern.ch/twiki/bin/view/Atlas/WorkBookAthenaFramework
24. GEANT tool homepage. http://wwwasd.web.cern.ch/wwwasd/geant/

Appendix A. Test queries.

Common for all tests

```
/* random numbers used in tests A and C */
create function randints(Integer)->Integer;
set randints(i) = r from Integer i, Integer r where i = iota (0, 999) and r =
rand(10000);

/* random numbers used in test B */
create function randcont(Integer i key)->Integer r;
set randcont(i) = r from Integer i, Integer r
       where i = iota (0, 4999) and
             r = rand(99999);

create function randentries(Integer i key)->Integer r;
set randentries(i) = r from Integer i, Integer r
       where i = iota (0, 4999) and
             r = rand(99999);

/* any big number can be used instead of 99999 */

row_setcachesize(0); /* to see performance without cache comment out to see performance
using cache */
```

Test A

```
declare RootContainer :c;

set :c = openContainer("dc2.003007.evgen.A1_z_ee._00092.pool.root", "",
"POOLContainer_McEventCollection");
set expandable(getType("HepMC::GenVertex"), "m_event") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_in") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_out") = 0;
set expandable(getType("HepMC::Flow"), "m_particle_owner") = 0;

create function testA(RootContainer c) -> Integer
       as select count (
              select get(c, randints(i)) from Integer i
          );
```

testA(:c); /* only this query was measured */

```
closeContainer(:c);
```

Test B

```
set :cs = /* automatically generated code to open 299 data files */
{
```

```
openContainer('rome/rome.004101.recov10.T2_McAtNLO_top500._00001.AOD.pool.root',
'','POOLContainer_ElectronContainer'),
openContainer('rome/rome.004101.recov10.T2_McAtNLO_top500._00004.AOD.pool.root',
'','POOLContainer_ElectronContainer'),
...
openContainer('rome/rome.004101.recov10.T2_McAtNLO_top500._00299.AOD.pool.root',
'','POOLContainer_ElectronContainer')
};

create function testB(Vector vc)->Integer as
select count (
       select get(c, cast(mod(randentries(i), entryCount(c)) as integer)) from Integer i,
RootContainer c where c = cast (vc[cast(mod(randcont(i),count(vc)) as integer)] as
RootContainer) and i = iota(0, 4999)
);
```

testB(:cs); /* only this query was measured */

```
for each RootContainer c closeContainer(c);
```

Test C

```
create function particle(struct s, integer i)->struct as select
getStructMember(getStructElement(getVectorMember(getStructElement(getVectorMember(s,
"m_pCont"), 0), "m_particle_barcodes"), i), "second");

create function length(real dx, real dy, real dz)->real as select sqrt (dx*dx + dy*dy
+dz*dz);

create function countOfLengths(RootContainer c) -> integer as
select count(
   select length(cast (getMember(mom, "dx") as real), cast(getMember(mom, "dy") as real),
cast (getMember(mom, "dz") as real))
   from struct s, struct mom, integer i, integer j, struct rec
   where
      rec = cast (get(c, randints(i)) as struct) and
      s = particle(rec, j) and
      mom = cast (getMember(cast(getMember(s, "m_momentum") as struct), "pp") as struct)
      and i = iota(0, 999)
);

declare RootContainer :c;
set :c = openContainer("../data/dc2.003007.evgen.A1_z_ee._00092.pool.root", "",
"POOLContainer_McEventCollection");
set expandable(getType("HepMC::GenVertex"), "m_event") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_in") = 0;
set expandable(getType("HepMC::GenVertex"), "m_particles_out") = 0;
set expandable(getType("HepMC::Flow"), "m_particle_owner") = 0;
```

testC(:c); /* only this query was measured */

```
closeContainer(:c);
```