Event data storage and management in STAR

V. Perevoztchikov[1]

Brookhaven National Laboratory,USA

Abstract

The Solenoidal Tracker At RHIC (STAR) is a large acceptance collider detector, commissioned for operation at Brookhaven National Laboratory in 1999.

STAR is designed to measure the momentum and identify several thousands of particles per event. About 300 Terabytes of data will be generated each year.

To handle such a huge amount of data, sophisticated data structures and associated tools were developed.

The main features of these data structures are:

- Data structure is a complicated but flexible set of C++ objects;
- All data objects are persistent. We do not maintain separate transient and persistent data structures;
- Persistence of data structure is based on ROOT[2] I/O implementation;
- Part of ROOT I/O was modified to meet STAR requirements;
- The modification of ROOT I/O allows to support schema evolution of STAR data structures. Automatic tools allow reading of old data into new environments.

In this paper we present our experience with maintenance of big and complicated OO data structures, especially concerning schema evolution.

Keywords: I/O,CHEP,ROOT.event

1 Introduction

In each HEP experiment I/O is a serious problem. In the era of C++, with much more sophisticated data organisation, the I/O problem becomes even more important and complicated. The STAR[1] experiment is not an exception. Our solution of this problem is based on the following approaches:

- All the persistent data is organised as a set of named components;
- Each component contains a tree of named data-sets;
- By default each branch is written in a separate file;
- File consists of keyed records where each record contains one component of one event;
- Different records with different components, with the same key (Run/Event) represent one full event;
- Each production stage adds one or more components;
- I/O implementation is based on ROOT[2] I/O;
- Schema Evolution is implemented on the base of our modifications of ROOT I/O.

All the above allows us to combine flexibility, power, and hopefully, user friendliness.

2 STAR I/O components

The STAR event is large and complex. Different parts of it are created in different processing stages. To ease management and increase storage flexibility we decided to split the event into

more simple parts – components.

An I/O component is a part of the event produced in one stage of event processing. Each new stage, for instance kinematic analysis, does not modify the existing components, but adds one or more new ones. As a result, a fully reconstructed event consists of a set of components (branches). The main features are:

- Each component lives in a separate file;
- Each offline stage reads existing components and creates new ones, without modifying or extending old files;
- The size of a full event in STAR is very big ($15 - 20MB$ in raw form). Separation of components allows to keep at least 50 events in one file, with the 1GB limit per a file;
- It is easy to add a new offline stage, without reorganisation of existing data;
- It is easy to reprocess events from any stage;
- Any application can read only needed files;
- The most frequently used files / components can reside on disks, the others on tape.

One component / file consists of a set of keyed records. Direct access for any record is allowed. The keys for different branches of the same event are the same and based on Run / Event numbers. This allows to construct one full event, reading several files in parallel.

Records, in turn, contain a tree of named datasets[3]. The structure of the tree is not predefined. Addition or removal of a dataset from the named tree does not affect the behaviour of modules which do not use them.

There are several special service components:

hist component - contains named tree of histograms, filled by different modules during processing.

runco component - contains named tree of Run/Control parameters used in reconstruction;

tagdb component - contains named tree of physical tags, defined in different modules. This component is used for filling of the STAR TagDB.

These components contain only one record per file and are filled at the end of processing sets of events.

A group of files / components with the same set of events organises a ”family” of files. Each file, in addition to its component, keeps information about the all other components existing at that time. Thus the last file in the production chain keeps information about the all produced components and files. When an application needs to use several components, it is enough to open one file and select needed component names. All the needed files from the ”family” will be opened automatically.

Such component organisation looks rather complicated, but for a huge event size and a complex processing chain, it allows to split complex events into relatively simple parts to simplify management. In an environment of larger numbers of smaller events a simpler approach could be appropriate.

3 ROOT I/O in STAR

ROOT I/O was chosen as the main mechanism of persistence in STAR. The main power of ROOT I/O is removing the artificial separation between transient and persistent data objects. With some reasonable limitations, the user is free to develop complex data objects without concern for the I/O implementation, and – importantly – without building dependence on the I/O scheme used into the data structures. ROOT automatically creates a streamer method for user defined classes and provides persistence of the object. For some rare, more complicated, objects, the user can write this streamer method himself.

There are special I/O objects in ROOT which are descendants of PAW NTuples. These objects (TTree/TBranch) are very convenient for NTuple-like objects in physics analysis. But for really complicated data objects they are too simple. To support persistence in STAR, we use ROOT I/O directly. To support STAR I/O component schema we developed StTree/StBranch classes which are similar to TTree/TBranch in ROOT, but with functionality specific to our needs. These classes are not intended to replace the related ROOT classes. ROOT TTree/TBranch objects are still used for physics analysis in STAR.

There are three stages of class development:

- User develops his class;
- ROOT reads class definition and creates codes for corresponding streamer method;
- Compilation of user and ROOT generated code and creation of shared library.

However when the user modifies the definition of his class and ROOT rewrites the corresponding streamer method, then previously written data becomes inaccessible. ROOT does not yet support schema evolution. Schema evolution aside, ROOT I/O is completely sufficient for us.

4 Schema evolution

Complete schema evolution is an unachievable goal, but schema evolution with some limitations is possible. These limitations must of course be reasonable. There are two solutions:

- Reading the old formatted data into memory and then the new application deals with the old data;
- Reading and converting the old format into the new one and then the new application deals with the new format.

The first approach was used in ZEBRA. ZEBRA can read any ZEBRA file and it is the problem of the application to work with the old format. This approach is completely impossible in C++. There is no way to create an old C++ object when the new one is declared. So, we must somehow convert the old data into the new format. To achieve this we modified the ROOT disk format by splitting the whole task of writing into numerous but simple ”atomic” subtasks.

Writing:

- Each object is written separately. All its members are written close to each other, i.e. we do not follow pointers to other objects inside the current one. The writing of these objects is delayed. This allows to easily skip unknown or unneeded objects;
- Each member which is a C++ class also is written separately;
- Streamer of an object is split by ”atomic” actions. An action is applied to one member. Each action has:
  - Numeric code related to the type of action. For example: fundamental type of member, pointer to fundamental type, ROOT object, pointer to ROOT object, general C++ object, etc..., have different code numbers.
  - Type name of member;
  - Name of member;
  - Number of repetitions in the case of array;
- The description of these ”atomic” actions is stored into the file together with data. It is not the description of written classes; it is the description of streamers, the description of how the objects were written.

When the output format is formalised in such a way we can compare the streamer descriptions of old and new data.

Reading:

- Read the streamer descriptions of old classes;

- Got an old object. If such a new class is known, create it. If not, skip the old class;
- Got an old ”atom”. If we have the new ”atom” with the same code, type and name, we fill it. If not, skip it.
- Some members of the new object could not be filled. It is the responsibility of the class designer to provide default filling of it.
- Etc.

After conversion of old objects into new ones, an application should know what to do if some members of the objects were not filled. But this is a problem of application schema evolution. I/O schema evolution is solved.

5 Conclusion

- STAR I/O based on component approach and ROOT I/O was implemented. It is has been working for one year;
- ROOT I/O was rewritten and schema evolution implemented. It is in testing stage now. Performance:
  - Schema evolution off:
    - Writing is the same speed as standard ROOT;
    - Reading is 30% faster than standard ROOT;
  - Schema evolution on: the same speed as standard ROOT.

References

1 J.W. Harris at al., “Conceptual Design report for Solenoidal Tracker at RHIC”, (The STAR Collaboration),June 1992, LBL Pub-5347.

2 Rene Brun,Nenad Buncic,Valery Fine, Fons Rademakers. ”ROOT: An Object-Oriented Framework”, ”International Workshop AIHEPN’96”, Aug, 1996, Laussane.

3 V.Fine. ”STAR C++ ROOT-based class library”, ”US HENP ROOT Users Workshop” March 23-25th, 1999, FNAL.