Event data storage and management in STAR

V. Perevoztchikov
Brookhaven National Laboratory,USA

Introduction

The Solenoidal Tracker At RHIC (STAR) is a large acceptance collider detector. 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 it, sophisticated data structure was developed. The main features are:

- All the persistent data is organized as a set of named components;
- All data objects are persistent. No separation between transient and persistent data structures;
- Persistence is based on ROOT I/O;
- Automatic schema evolution is implemented. It was developed on the base of non automatic ROOT schema evolution ;
- The system is working.

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 - *named components*

- 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) . 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 reorganization 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.

STAR I/O components (continued)

- One *component* consists of a set of keyed records. These keys are 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;
- The tree structure is not predefined. Addition or removal the tree does not affect the behaviour of modules which do not use them.

All components are born equal. But some of them are more equal than others. They contain only one record per file.

- ***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 to fill the STAR TagDB

STAR I/O components (continued)

- A group of *files/components* with the same set of events organizes 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 production chain keeps information about the all produced components and files.
- It is enough to open any file and select needed component names. All the needed files from the *"family"* will be opened automatically.

Such component organization looks rather complicated, but for a huge event size and a complex processing chain, it allows to split complex event into relatively simple parts to simplify management.

In an environment of larger numbers of smaller events a simpler approach could be appropriate.

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 :

- No artificial separation between transient and persistent data model.
- User is free to develop complex data objects without concern for the I/O implementation, and -- **importantly** -- without building dependence on the used I/O scheme;
- Automatic creation of a *streamer* method for user defined classes, which provides persistence of the object;
- For special, more complicated, objects, user still can write this *streamer* method himself.

STAR I/O classes

The component organization of STAR I/O is supported by STAR I/O classes: StTree,StBranch, StIOEvent and StFile (no relation to ROOT TTree and Tbranch classes).

StTree - container of *components*;

StBranch - representation of STAR I/O *component*;

StIOEvent - ROOT I/O connection;

StFile - container of files.

These classes perform I/O, add, fill, update of files/*components*

They are heavily based on ROOT environment and work well.

However when 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 *automatic schema evolution*.

Schema evolution aside, ROOT I/O is completely sufficient for us.

Automatic Schema evolution

Complete schema evolution is an unachievable goal, but schema evolution with some limitations is possible. The limitations must 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**.

Automatic Schema evolution(continued)

To achieve this, we have modified the ROOT disk format by splitting the whole task of writing into numerous, but simple "*atomic*" subtasks.

- Each object is written separately. All its members are written close to each other;
- Pointers to object are not followed immediately. Writing of these objects is delayed. This allows to skip unknown or unneeded object;
- Member which is a C++ class is written as a separate object;
- Streamer of an object is splited by "*atomic*" actions. An action is applied to one member. Each action described by:
  - Numeric code related to the kind of action. For example:
    - member of fundamental type;
    - pointer to fundamental type;
    - C++ object;
    - pointer to C++ object.
    - Etc...

Automatic Schema evolution(continued)

- 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 formalized 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 class is known, **create** it. If not, **skip** object;
- Got an old "*atom*". If we have the new *"atom"* of the same kind, type and name, **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 them.

After conversion, an application should deal, with not filled members. But this is a problem of **application schema evolution**. **I/O schema evolution is solved**.

Conclusions

- 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 modified and automatic schema evolution implemented. It is in testing stage now. Performance:
  - The same file size as for standard ROOT;
  - The same speed as standard ROOT I/O.

Current status:

- Components and ROOT I/O has been working for one year;
- Codes of modified ROOT I/O and automatic schema evolution are ready and will be tested in real production.