Schema Evolution Implementation in ROOT I/O

V. Perevoztchikov
Brookhaven National Laboratory,USA

STAR

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**.

Modified ROOT I/O Format

Modified ROOT I/O format is based on the last version of standard ROOT I/O. There is no *ideological* difference, it is slightly different implementation. The main feature is the possibility to skip not only object but any member of object. It is essential for schema evolution.

- Each object has a header containing flag,classid and objectsize ;
  - Short header - one 32 bits word (Classid < 1K && ObjectSize< 1M);
  - Long header - two 32 bits words. (Classid > 1K || ObjectSize> 1M);
- Object is written continuously, pointers are not followed immediately.
  - Simple members written immediately;
  - TObject* & TObject: buffer offset of object is written. Object itself is written separately and schema evolution is applied for it recursively;
  - General C++ class: written immediately preceded by its size;
- Reference pointers either zero or offset of object in buffer. This is a new feature.;
- List of used classes is written at the end of the record.

Automatically Generated Streamer

The new automatically generated streamer method is more complicated than a standard one. An additional communication with TBuffer class is developed.

- At the beginning, Streamer asked TBuffer *is class modified*? If not, it works as usual;
- Before reading of a member streamer requests TBuffer permission to read it. If permission is granted, **reading**, if not, **next** member;
- When Streamer returns, it could be called again, to read some skipped members. It could be happened if the order of members was changed

When it works?

- New member added;
- Old member removed;
- Type of member changed, ie. Int to float,int to short, etc…;
- Array size changed;
- Definition of class member changed;

Streamer example

```
void TLorentzVector::Streamer(TBuffer &R__b) {
//           Stream an object of class TLorentzVector.
  void (*R__bs)(TObject *,TBuffer*);
  Version_t R__v = 0;
  if (R__b.IsReading()) {
    int R__Comp  = R__b.DoIt();
    if (R__Comp || R__b.DoIt(10)) R__v = R__b.ReadVersion();
    TObject::Streamer(R__b);
    if (R__Comp || R__b.DoIt(40,"fX","double",8)) R__b >> fX;
    if (R__Comp || R__b.DoIt(40,"fY","double",8)) R__b >> fY;
    if (R__Comp || R__b.DoIt(40,"fZ","double",8))  R__b >> fZ;
    if (R__Comp || R__b.DoIt(40,"fE","double",8))  R__b >> fE;
  } else {
  // Writing part is skipped
}}
```

TStreamer class

To perform schema evolution, old and new classes should be compared;
Class TStreamer keeps information how class was written. Based on this,
TBuffer::DoIt method allows or disallows to Streamer to read current part of TBuffer.

- One instance of TStreamer related to one instance of Tclass;
- It keeps old class check sum. Comparing this with current class, TStreamer makes decision was class modified or not;
- It keeps information about all “*atomic*” operations of old class;
- Above information is produced by **exactly the same** code by which automatic Streamer was generated;
- Hash list of TStreamer’s belongs to TFile and saved during Close()
- When TFile is opened this information restored and acceptable to TBuffer

Modified ROOT classes

- **TStreamer** - new class introduced;
- **TClass** - GetClassID method, returns class check sum;
- **TFile** - add TStreamer hash list;
- **rootcint** - new automatic Streamer generated;
- **TBuffer** - big modifications;
- **TKey** - minor modifications;

Conclusions

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

Current status:

- Codes of modified ROOT I/O and automatic schema evolution are ready and should be tested in real production.

Future

Future of STAR-like ROOT schema evolution , as usual for any future, is unclear.

It could be 3 scenarios:

- The best one: our automatic schema evolution will be accepted by ROOT framework;
- It will not be accepted. Then we introduce StTBuffer,StTFile, etc… inherited from standard ROOT classes and will be used in STAR. It is not convenient, but possible solution. (As Rene told - schism);
- Somebody will implement better schema evolution and we will accept it