Data Persistency Solution for LHCb

G. Barrand[1], *I. Belyaev*[2], *P. Binko*[3], *M. Cattaneo*[3], *R. Chytracek*[3], *G. Corti*[3], *M. Frank*[3], *G. Gracia*[3], *J. Harvey*[3], *E. van Herwijnen*[3], *B. Jost*[3], *I. Last*[3], *P. Maley*[3], *P. Mato*[3], *S. Probst*[3], *F. Ranjard*[3], *A. Tsaregorodtsev*[3]

[1] Laboratoire de l'Accélérateur Linéaire (LAL), Orsay, France
[2] Institute for Theoretical and Experimental Physics (ITEP), Moskva, Russia
[3] European Laboratory for Particle Physics (CERN), Genève, Switzerland

Abstract

The GAUDI software architecture, designed in the context of the LHCb experiment, maintains separate and distinct descriptions of the transient and persistent representations of the data objects. One of the motivations for this approach has been the requirement for a multi-technology persistency solution such that the best-adapted technology can be used for each category of data: raw event data, reconstructed event data, summary event data, detector description, event catalogues, statistical data, etc. This approach has also allowed us to evolve smoothly with time from current legacy persistent data, to more sophisticated solutions that will appear in the future, such as object database management systems. A simple generic mechanism has been developed for converting data between their transient and persistent representations and for resolving, on demand, associations through different persistent solutions. We intend to use this mechanism for ROOT, Objectivity/DB and ZEBRA (legacy) data. We describe the basic concepts, as well as the more detailed design issues of this multi-technology persistent solution. The performance and practical experience in its use will also be presented.

Keywords: LHCb, GAUDI, data store, persistency

1 Introduction

LHCb is a dedicated B-physics experiment being prepared at the LHC collider at CERN [1]. LHCb will produce various types of data which can be categorized according to their nature and role during data processing. All these data need to be processed by various software applications developed around an object-oriented software architecture that has been recently designed; the GAUDI architecture [2]. One of the most important design features of such an architecture is the way data persistency issues are handled. The reasons for the approach taken in GAUDI with respect to data persistency are described in some detail in the following sections.

2 Multi-technology persistency solution

Several important considerations have led us to conclude that our software architecture should support in a transparent way the use of different persistency solutions for managing the various types of data that must be treated in our data processing applications. Firstly the volumes for the different data categories vary by many orders of magnitude. The event data from the different processing stages (raw data, reconstructed data and summary data) account for roughly 0.5 PB/year, while the detector data and event catalogues demand a few TB/year. Configuration and bookkeeping data will require only a few MB per year. Different access patterns are typical for these different data stores e.g. write-once/read-many for event data, read and write many for other data, sequential access, random access, etc. In addition, the software framework should support storage and access to legacy data. The simulated data currently used for the detector design studies are stored using ZEBRA [3] and the data produced in some of the test beams are made persistent using the ROOT framework [4]. For these reasons we believe that a single persistency technology may not be optimal in all cases. The GAUDI software architecture has been designed such that the best-adapted technology can be used for each category of data and allow a smooth integration of the existing data. This approach will also allow us to evolve smoothly with time to more appropriate solutions as they appear in the future.

3 The transient data store

The GAUDI design feature that helps to satisfy the goals mentioned in the previous section is the creation of two representations for data objects, the transient and the persistent. The existence of a transient store helps to minimise coupling between algorithm objects and data objects. This approach was inspired by the work done in the BaBar experiment [5]. An algorithm can deposit some piece of data into the transient store, and these data can be picked up later by other algorithms for further processing without knowing how they were created. This architecture conforms to the ”black-board” architectural style, in which the transient store fulfils the role of the blackboard.
The transient data store also serves as an intermediate buffer for any type of data conversion, in particular the conversion into persistent objects. Thus data can have one transient representation and zero or more persistent representations. Having different representations allows for different optimization criteria:

- Persistent data are usually optimised in terms of their storage allocation. This typically involves the use of data compression and the minimisation of links between objects.
- Transient data can be organised to optimise the execution of algorithms, for example by duplicating information if needed (caching).

The organisation of the data within the transient data stores is ”tree-like”, similar to a Unix file system. This allows data items that are logically related, such as Monte Carlo ”truth” information, to be structured and grouped at run-time. Each node in the tree may contain data members, but also other nodes containing further groups of data members (Figure 1). As in a directory structure, each node is the *owner* of everything below it and will delete all these items when it gets deleted. In general, object-oriented data models do not map onto a tree structure. Thus, mesh-like object associations have been implemented using symbolic links (again inspired from the Unix file system) in which the node does not acquire ownership of the referenced item. Data stores are pop-

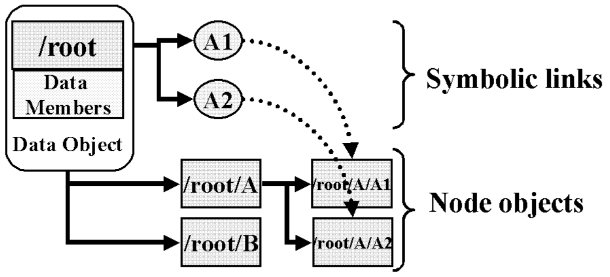

Figure 1: A node in the transient store.

ulated only on demand of *Algorithms* in order to minimise the overhead of creating unnecessary objects. Stores can be browsed for the existence of an object or to selectively build collections of objects that will be migrated to persistent storage.

4 Data access and data conversion

There are several options for maintaining both data representations. One is to describe the user-data types within the persistent storage framework (meta-data) and have utilities able to automatically create both representations using this meta-data. This approach is elegant and relatively easy for basic data types, but is complicated when converting objects with many relationships.
Another possibility is to code the conversion specifically for each data type, and this is the approach chosen in GAUDI. A *Converter*, with a common interface, is called whenever an object needs to be created in another representation. Each *Converter* is able to convert objects of one type (given by a class identifier) between the transient and one other representation (given by the representation type).
The *Converter* can perform complicated operations, such as the combination of many small transient objects into a single object in order to minimise overhead in storage space and I/O. When converted to the transient representation, the persistent representation is expanded to the individual objects. This flexibility is only possible if the code is specifically written.
Every request for an object from the data service invokes the sequence shown in Figure 2:

- The data service searches the data store for the requested object. If the object exists, a reference is returned and the sequence ends.
- Otherwise the request is forwarded to the persistency service. The persistency service dispatches the request to the appropriate conversion service capable of handling the specified storage technology. The selected conversion service uses a set of data converters and forwards the request to the proper converter.
- The converter accesses the persistent data store, creates the requested transient object and returns it to the conversion service.
- The conversion service registers the object with the data store and the data service subsequently returns a reference to the client.

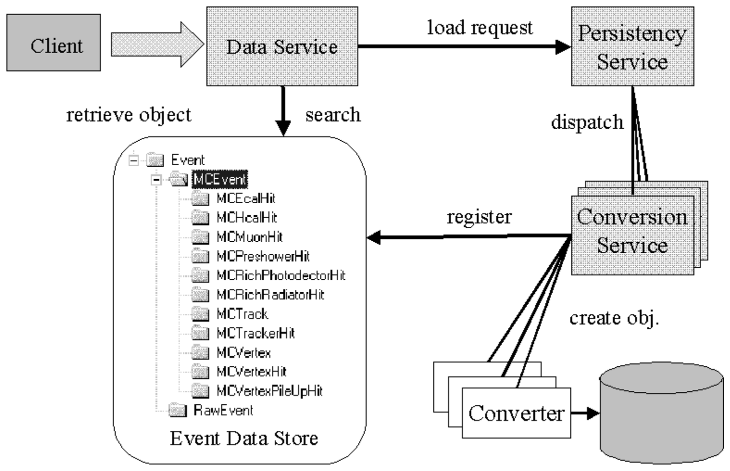

Figure 2: Action sequence for loading an object from the persistent medium.

When making objects persistent, the calling sequence is as follows:

- An agent traverses the data tree using the data service collecting references to objects that are to be made persistent.
- The collected references are passed to the persistency service, which dispatches each object referenced to the appropriate data converter.

- Each converter allocates persistent storage on the requested medium and fills the allocated area according to the transient object.
- As a last step all persistent references are filled and the updated persistent objects stored.

Note that this mechanism can also be applied to conversions to other data representations, such as those used in visualisation.

5 The generic persistent model

Traditionally HEP data was accessed through sequential files. The file was organised in logical records representing one event partitioned into banks. The drawback of this organization is the difficult to access banks from previous processing steps, for example to rerun a reconstruction algorithm while analysing summary data.
OODBMS and RDBMS technologies allow for this type of random access. Storing primitive properties of an object with these technologies is simple, but it is difficult to store references to other objects, as these pointers are only valid in the current address space and need special care.
OO databases solve this problem by replacing the reference with an object identifier (OID) which allows the ODBMS engine to locate the persistent representation of the object. In addition, the ODBMS engine manages the dynamic behaviour (methods, polymorphism, inheritance) of the objects delivered to the user by setting up the proper function table. Unfortunately in existing ODBMS technologies this mechanism is implementation specific, and does not allow reference to objects outside the current database engine.
A generic persistent model should allow for the following actions, namely to:

- select the correct storage engine to create the object representation.
- locate the object on the storage medium.
- validate the object properties according to the persistent representation.
- handle the object's dynamic behaviour by setting up the proper function table.
- rreserve the transient data store structure.

Our schema assumes that most database technologies are based on files or logical files. Internally these files are partitioned into containers (database containers in Objectivity/DB, ”Root trees” for ROOT I/O, tables for an RDMS)and objects populating these containers. The implementation-specific OID must be replaced by an extended object identifier (XID) (see Figure 3) containing all relevant information.

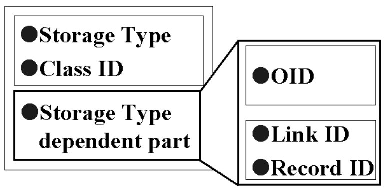

Figure 3: Layout of the extended object identifier (XID).

If the XID represents an object in an ODBMS, the storage-dependent part is identical to the OID. Otherwise all information required to locate the database file, the container/table and the corresponding record must be determined from this part. Object type, Storage type, database name, container name, object name/path in the transient store and the object identifier within the container (record ID) form a universal address, which allows data items to be addressed in nearly any known technology. To minimise persistent storage, this address is split, with database name, container name and object name path stored in a separate lookup table, identified by a primary key, the link ID (see Table I). When writing, the look-up table needs to be updated whenever a new link that is to be made persistent occurs in the object model.

Storage Type	Link Context information		
<generic>	File/DB name	Container Name	Object path
ZEBRA	ZEBRA file name	Bank name	Object path
ROOT I/O	ROOT file name	Tree name	Object path
Objectivity/DB	Database name	Container name	Object path
ODBC / MS Jet	Database name	Table name	Object path

Table I: Layout of the universal container address in the link table. The role of the database name and the container name depends on the persistent technology.

The organisation of the data in the transient store uses ”leaf” objects and symbolic links. In the persistent world the handling of references to leaf objects and of symbolic links is basically the same, both condense to a set of XIDs within the persistent representation. However, when reading, the interpretation of an XID is different:

- An XID representing a leaf object in the persistent world must be converted into a leaf stub in the transient data store. The stub contains all information needed to later load the object from the persistent medium. The stubs for the subsequent layer are created when a node is converted.
- An XID representing a symbolic link translates to the full path of the addressed object within the transient store. The object path is part of the table's link information, which is used later to partially load the corresponding branch containing the referenced object.

The information is sufficient to determine the path of the linked object within the transient store and allows the construction of a simple load-on-demand mechanism for embedded relationships, for example using smart pointers.

6 Experience with various persistency solutions

The model described above has been implemented for several engines. In LHCb, ZEBRA data are read only and will not be written using this technique. ROOT I/O is currently the preferred solution to write user data. Other existing prototypes use Objectivity/DB and ODBC. Accessing relational databases via ODBC could be used for small data selections.
The system has been tested so far on a rather small scale and performs well. The overhead introduced by separating transient and persistent data has been measured and the performance penalty found to be acceptably small when compared to overall cpu usage.

References

1 S.Amato et al., LHCb Technical proposal, CERN/LHCC 98-4.
2 G.Barrand et al., GAUDI - The Software Architecture and Framework for building LHCb Data Processing Applications, CHEP 2000, Proceedings.
3 R.Brun, ZEBRA - Reference Manual - RZ Random Access Package, Program Library Q100. CERN, 1991.
4 R.Brun and F.Rademakers, ROOT - An Object Oriented Data Analysis Framework, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also `http://root.cern.ch`.
5 S.Gowdy et al., Hiding Persistency when using the BaBar Database, CHEP 1998, Proceedings.