POOL DEVELOPMENT STATUS AND PLANS

I. Papadopoulos, R. Chytracek*, D. Düllmann, M. Frank, M. Girone*, G. Govi*, J.T. Moscicki*,
H. Schmücker (CERN, 1211 Geneva, Switzerland)
K. Karr#, D. Malon#, A. Vaniachine# (Argonne National Laboratory, IL 60439, USA)
T. Barrass (University of Bristol, BS8 1TL, UK)
W. Tanenbaum (Fermi National Accelerator Laboratory, Batavia, IL 60510, USA)
C. Cioffi (University of Oxford, Oxford, OX13NP, UK)
Z. Xie (Princeton University, NJ 08544, USA)

Abstract

The LCG POOL project [1] is now entering the third year of active development. The basic functionality of the project has been provided but some functional extensions will move into the POOL system this year. This paper summarizes the main functionality provided by POOL, which is used in physics productions today. It also presents the design and implementation of the main new interfaces and components planned such as the POOL RDBMS abstraction layer and the RDBMS based Storage Manager back-end..

INTRODUCTION

POOL is one of the two parts of the LCG persistency framework; the other one is the ConditionsDB project. POOL provides technologically neutral object persistency with navigational capabilities integrating object streaming and relational database technologies. The high level design and architecture of the POOL system is described in [2].

POOL has been established as the baseline technology for the software object storage in three LHC experiments (ATLAS, CMS, LHCb) after its successful integration into their software frameworks [3]. Its current functionality has been largely validated by the experiments during this year's data challenges [4].

The main development efforts of the POOL team are currently focused towards the definition and implementation of a technologically neutral mechanism for accessing relational databases. One of the ultimate aims of this new development line is the extension of the object storage capabilities of POOL in order to be able to use relational database technologies, complementing the usage of the existing object streaming technology (ROOT I/O) [5].

At the same time the POOL team is working in collaboration with the ROOT developers to ensure a smooth transition towards adopting ROOT version 4 in a way that backwards compatibility will be ensured.

CURRENT STATUS

POOL has entered its third year of development. During the first two years the project team was focused towards following the proposed work plan. Our team has managed to meet the rather aggressive time requirements for producing the software deliverables at the quality level which was required in order for POOL to be part of the production software of the ATLAS, CMS and LHCb experiments.

In order to support this year's data challenges which have generated a volume of ~400TB the project team had to shift focus from pure development to user support, deployment and maintenance. To this end several developers have placed their effort into experiment software integration or back-end services. This strategic decision, which was meant to insure proper coupling between software and deployment, has affected the available development manpower with the task profile changing from design and debugging to user support and re-engineering. There is still though the need to maintain stable and focused manpower from CERN and the experiments; both parties have confirmed their continuous commitment to the project.

INCREMENTAL DEVELOPMENTS

Migration to ROOT version 4

This year the ROOT team released version 4 of their software framework. Since the I/O part of it is used as the main technology for object streaming in POOL, there has been a need for POOL to migrate to this version, marking the start of the POOL 2.0 development line.

Migrating to ROOT 4 is not only required on the basis of the configuration issues that arise if one considers that the clients of the LCG software may use ROOT also through paths not involving the POOL software components. ROOT 4 offers the advantages of automatic schema evolution and simplified streaming of the standard C++ library containers.

The main challenge in this effort is to ensure backwards compatibility for POOL 1.x (ROOT 3.x) files. This issue is being resolved through the close collaboration between the POOL and ROOT teams, which try to agree on the file format of the ROOT 4 files containing standard C++ library containers. At the same time there is an undergoing validation process by the experiments of the POOL 2.0 pre-releases.

The POOL team will be releasing two branches, one based on ROOT 3 and another one based on ROOT 4, until POOL 2 is fully certified.

*funded by Particle Physics and Astronomy Research Council, UK.
#work supported in part by the U.S. Department of Energy, Division of High Energy Physics, under contract W31-109-Eng-38

File Catalog deployment

This year's data challenge productions have been heavily based mainly on XML and grid catalog implementations. For the latter several weaknesses have been revealed over which POOL has little control. At the same time several new or enhanced catalogs are being developed. Moreover, changes in the computing models of the experiments need to be taken into account.

POOL is trying to generalize from specific implementations and to provide an open interface to accommodate upcoming components. To this end the File Catalog interfaces are being redesigned to achieve a clear split between user- and developer-level interfaces, between catalog management and functionality and between meta-data handling and file name registration and lookup.

The new interface design will ease the development of adaptors matching the POOL catalog interfaces and the API of the underlying grid services. A testing suite based purely on the POOL File Catalog interfaces will be used by developers of new implementations to validate and benchmark their components.

Collection Catalogs

There are currently several implementations of the POOL Collection interfaces. These are either implicit, implemented directly at the Storage Manager level, or explicit implemented using ROOT trees or MySQL tables. In response to experiment requests, cataloguing of explicit collections has been recently provided. Collection catalogs are similar to file catalogs, where the entries are named collections instead of files. For the first implementation of the collection catalogs we have reused the existing file catalog implementations and command-line tools.

Further development in the area of the Collections needs concrete input from the analysis models of the experiments. We are expecting that the experience gained from the analysis parts of this year's data challenges will provide us with the desired feedback.

A RELATIONAL BACK-END FOR POOL

Motivation and goals

The first discussions on a relational back-end for POOL started towards the fall of 2003 between the POOL team and the LHC experiments. There were two main physics use cases that had to be addressed. The first one is related to the ConditionsDB project [6]. It had already become evident to everybody that the data payload for the conditions objects should be handled by POOL, which already provides a general object storage mechanism, while keeping the intervals of validity in a relational database. In order to avoid having to manage two types of storage media when storing conditions objects, POOL had to provide a Storage Manager implementation based on the same relational database technology that is used for storing the intervals of validity.

The second use case arises from the fact that there configuration and detector control data that are written on-line directly to relational databases using native APIs or vendor-specific tools. Off-line reconstruction and analysis frameworks often require such data to be read in as software objects, which can be referenced by other reconstruction or analysis objects. An example would be a reconstructed event header pointing to objects holding information such as the beam luminosity or the detector layout corresponding to the time that the actual physics event took place. A relational back-end for the POOL Storage Manager would have to handle existing relational data which have to be presented as user-defined software objects.

Domain decomposition

During the first months of 2004, the use cases for the relational back-end have been formalized in a requirements document authored by members of the POOL team and representatives of the LHC experiments. The analysis of the requirements lead to the domain decomposition which is shown in Fig.1.

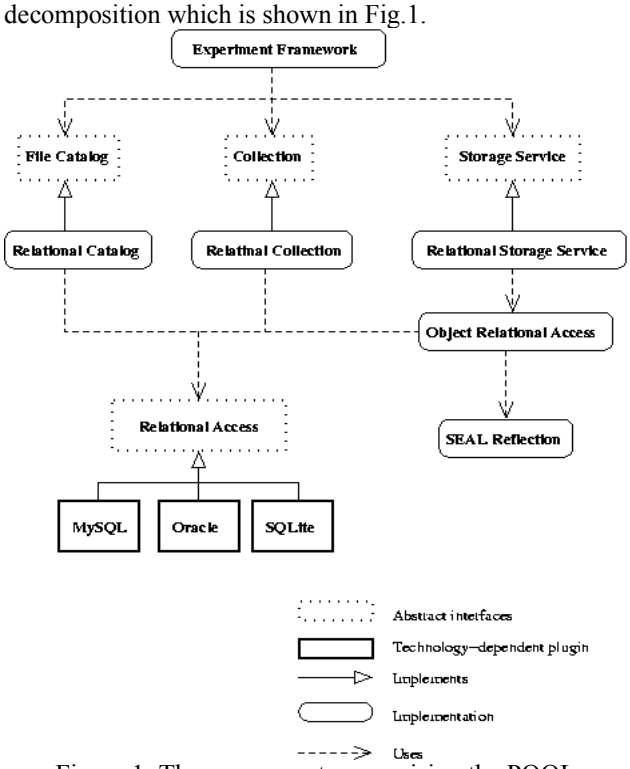

Figure 1: The components comprising the POOL relational back-end and their relation to the rest of the software system.

The POOL relational back-end comprises three main domains.

- The Relational Abstraction Layer (RAL), which is defines a technologically neutral API for accessing and manipulating data and schemas in relational databases.
- The Object-Relational mapping mechanism, which is responsible for transforming C++ object definitions to relational structures and vice-versa.
- The Relational Storage Service, which is an adapter implementing the POOL Storage Service interfaces in terms of the RAL and using the Object-Relational mapping mechanism.

The Relational Abstraction Layer

The RAL has been identified as the base domain for the whole relational back-end for several reasons. It is required in order to achieve vendor independence for the relational components of POOL (File Catalog, and Collections), the ConditionsDB and potentially user code of applications accessing relational data. Moreover, its introduction may address the problem of distributing data in RDBMS of different flavours.

The RAL abstract interfaces are defined in the *RelationalAccess* package. Their technology-specific realizations are implemented following the SEAL component model [7] as plug-in libraries. This architecture reduces the code maintenance effort for the relational components and allows for a better traceability of bugs. Moreover it minimizes the risk of binding to a particular RDBMS vendor. On the contrary, it allows the usage of multiple technologies in parallel. Applications which access relational databases through the RAL become automatically testing grounds for plug-ins of new RDBMS flavours.

The RAL interfaces allows a user to:

- describe or manipulate an existing schema, i.e. create and describe tables and indices, define and retrieve primary keys, unique, null and foreign key constraints;
- perform data manipulation, i.e. insert, delete and update rows in a table;
- perform queries involving one or more tables, supporting nested queries, limiting and ordering of the result set, client cache control and scrollable database cursors.

The handling and the description of the relational data is done using a simple key-value pair interface of the already existing POOL *AttributeList* package. The RAL API is a clean C++ interface with no SQL types involved. The only SQL fragments a user would ever have to provide is the WHERE and SET clauses in the data manipulation operations and the queries. The C++ to/from SQL type conversion is done implicitly through a type converter. Each technolgy implementation provides a default type mapping which is user customizable so that one could take advantage of vendor-specific SQL type extensions.

The encapsulation of the SQL types and syntax behind the C++ interfaces solves the problems which arise from the non-compliance of the various vendors to a common standard for some SQL operations such as table creation. It therefore shields the clients from the technology-specific software not only by eliminating compile-time dependencies but also semantically.

The choice of the specific plug-in which has to be loaded during run time is deduced from the technology field of the connection string which is provided by the user. This string should have the following format in order to be recognizable by the system:

technology[_protocol]://database[:port]/databaseSchema

No authentication parameters such as user name or password appear in such a string. The reason for this is that the connection string should be used to describe only the physical location of the data. Such strings are expected to be shared among different users or even stored as "physical file names" in the POOL file catalogs. The inclusion of the authentication parameters is therefore not appropriate.

A user authenticates oneself with the database either explicitly providing a user name and a password through the RAL API, or implicitly using an *Authentication Service*. Such a service provides the system with the necessary authentication parameters given a connection string. POOL has provided two implementations of the *IAuthenticationService* interface. One which reads the parameter values from two environment variables and another one which reads them from an XML file, where multiple connection strings and their corresponding authentication parameters are specified.

The RAL was first released with the POOL software in version 1.7. In this version two technology-specific plug-ins were provided as well: one for accessing Oracle databases and one for accessing SQLite files.

The Oracle [8] plug-in has been implemented using the Oracle Call Interface (OCI) client software. This choice was made mainly for two reasons. The first one was that we would like to profit from the performance advantages that this solution offers. The second is that being a C library we expect to be encountering less configuration problems whenever POOL is released with a new C++ compiler.

Since the first pre-releases of POOL 1.8 the Oracle plug-in is built against the Oracle Instant Client. It has been tested against 9i and 10g database servers. The software automatically detects the version of the database and in case of a 10g server it makes use of the recently introduced BINARY_FLOAT and BINARY_DOUBLE types which are stored as standard IEEE floating point numbers in the database.

SQLite [9] is a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine. It is file-based and therefore the consistency of concurrent accesses is guaranteed by the underlying file system.

As of the pre-releases of POOL 1.8 there is available a plug-in which serves accesses to MySQL [10] databases. This library has been implemented using the ODBC API. This means that the MyODBC driver is loaded during run time. The choice to use the ODBC API instead of the C native one was done for three reasons. The first one is to ensure smooth transition from the MySQL version 4.0 to version 4.1 and later to 5.0, where the native C API as well as the underlying semantics change considerably. The second reason is that the MyODBC driver exposes a more complete functionality, which had allowed almost the full implementation of the RAL interfaces. Finally, the third reason is that this plug-in can be eventually used to serve other RDBMS technologies for which a free ODBC driver exists.

The RAL has already been used within POOL to implement a Relational File Catalog. Some experiments have already integrated it in their frameworks and there are already experiment-specific applications accessing Oracle databases through the RAL of POOL.

Object storage using the RDBMS back-end

The second domain in the POOL relational back-end addresses the issues which emerge when a C++ class has to be mapped to a relational structure and vice-versa.

In the relational world tables play a similar role to that of the classes in the object world: they define how data are laid out in memory. Rows in a table can be thought of as the equivalent of objects of a class because they hold data of a well defined layout.

The first fundamental difference between objects and rows is that the former exhibit identity by construction while the latter by default not. Identity is necessary to uniquely and unambiguously address an object in a program in order to access its data. It is also the basis of every association between objects. To solve the problem of missing identity it is required that rows which are to be represented as objects should be in tables which define a primary key or a unique index.

The second difference between objects and rows has to do with the associations that may be established between two or more data sets. In the object world there are aggregations (associations realized as persistent references) and compositions. In the relational world the corresponding constructs are foreign key constraints. Object associations have a well defined directionality and multiplicity. On the other hand a table schema alone cannot determine unambiguously the directionality and the multiplicity implied by a foreign key constraint. It is up to the mapping process to resolve these ambiguities.

To illustrate how the mapping works let us assume that a user would like to store objects of the following C++ class:

```
class A {
   int m_i; float m_x; std::vector<B> m_b;
};
```

with

```
class B{
   float m_x; double m_y;
};
```

One of the possible mappings to a relational schema for this class is presented in Fig.2. The schema contains one table (T_A) for the *top-level* class A, and another one (T_A_M_B) to accommodate the values of the data member vector m_b. The primary key (ID) in T_A serves the role of the object identity. In the table T_A_M_B a foreign key constraint is defined. There is also a special column to hold the position of the elements inside the vector.

The *ObjectRelationalAccess* package of POOL provides the necessary software for generating mappings given a class. It allows a user to prepare the relational schema by creating or altering the relevant tables. While there are default rules for the mapping generation, a user could override them. This would be the case if one would like to generate object-relational mappings for existing data. POOL provides a tool which uses an XML file to steer the mapping generation, where the non-default rules are specified. The generated mapping is a hierarchical structure of elements describing the C++ types and names of the data members as well as the names of the associated columns and tables. The mapping hierarchy is versioned and can be stored in the database in three *hidden* tables.

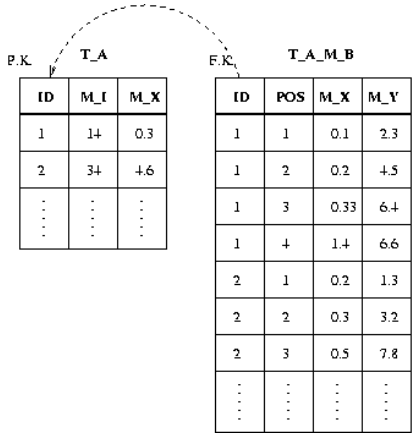

Figure 2: The relational schema corresponding to a mapped class.

Object storage and retrieval is performed with the guidance of the SEAL reflection information for the C++ class in question and the corresponding mapping element for this class. The version of the mapping ensures that simple schema evolution cases are handled automatically.

A POOL *container* of objects simply keeps the values of the primary key values and the mapping versions corresponding to an object whose data members are written to the relational tables. The POOL *RelationalStorageService* component, which will be released this year, will ensure that the full object I/O can be performed through the POOL framework in an identical -to the user- way with the existing object streaming to ROOT files.

ACKNOWLEDGEMENTS

The POOL team would like to thank the people working on the integration of POOL into the software frameworks of the LHC experiments for being the beta-testers of the various POOL components, contributing significantly to the quality of the our deliverables.

REFERENCES

[1] The POOL project http://pool.cern.ch/.
[2] D. Düllmann, “The LCG POOL Project – General Overview and Project Structure”, CHEP 2003 Proceedings, MOKT007.
[3] Giacomo Govi et al. “POOL integration into three experiment software frameworks”, these proceedings.
[4] Maria Girone et al. “Experience with POOL in the LCG data challenges of three LHC experiments”, these proceedings.
[5] The ROOT project http://root.cern.ch/.
[6] Andrea Valassi et al. “LCG Conditions Database project overview”, these proceedings.
[7] Radovan Chytracek et al. “The SEAL Component Model”, these proceedings.
[8] Oracle http://www.oracle.com/.
[9] SQLite http://www.sqlite.org/.