POOL Development Status and Production Experience

R. Chytracek, D. Düllmann, M. Frank, M. Girone, G. Govi, J. T. Moscicki, I. Papadopoulos, H. Schmuecker, K. Karr, D. Malon, A. Vaniachine, W. Tanenbaum, Z. Xie, T. Barrass, and C. Cioffi

***Abstract*—The pool of persistent objects for LHC (POOL) project, part of the large Hadron collider (LHC) computing grid (LCG), is now entering its third year of active development. POOL provides the baseline persistency framework for three LHC experiments. It is based on a strict component model, insulating experiment software from a variety of storage technologies. This paper gives a brief overview of the POOL architecture, its main design principles and the experience gained with integration into LHC experiment frameworks. It also presents recent developments in the POOL works areas of relational database abstraction and object storage into relational database management systems (RDBMS) systems.**

***Index Terms*—Data persistency, object streaming, plug-in architecture, relational database.**

I. Introduction

DATA processing at the large Hadron collider (LHC) imposes strict requirements on the computing models of all the LHC experiments.

The data rate and volumes foreseen are much larger than for previous experiments, and require a review of traditional approaches, which were typically based on explicit file handling by the end user. Furthermore, the long LHC project lifetime necessitates an increased focus on maintainability and change management for the core software, especially in the area of data handling. It has to be expected that during LHC project lifetime several major technology changes will take place and experiment data handling systems will have to adapt quickly to changes in the environment or physics research focus.

The POOL project [1] has been created in the context of the LCG Application Area to provide the LHC experiments with a common software framework for persisting data. The POOL data storage mechanism is intended to cope with the experiments' requirements by applying a flexible multitechnology data persistency mechanism.

Manuscript received November 11, 2004; revised May 24, 2005. The work has been supported in part by the U.S. Department of Energy, Division of High Energy Physics, under Contract W31-109-Eng-38.

R. Chytracek, D. Düllmann, M. Frank, M. Girone, G. Govi, J. T. Moscicki, I. Papadopoulos and H. Schmuecker are with CERN, 1211 Geneva 23, Switzerland.

K. Karr, D. Malona and A. Vaniachine are with Argonne National Laboratory, Argonne, IL 60439 USA.

W. Tanenbaum is with Fermi National Accelerator Laboratory, Batavia, IL 60510 USA.

Z. Xie is with Princeton University, Princeton, NJ 08544 USA.

T. Barrass is with University of Bristol, Bristol BS8 1TL, U.K.

C. Cioffi is with University of Oxford, Oxford, OX13NP, U.K.

Digital Object Identifier 10.1109/TNS.2005.860141

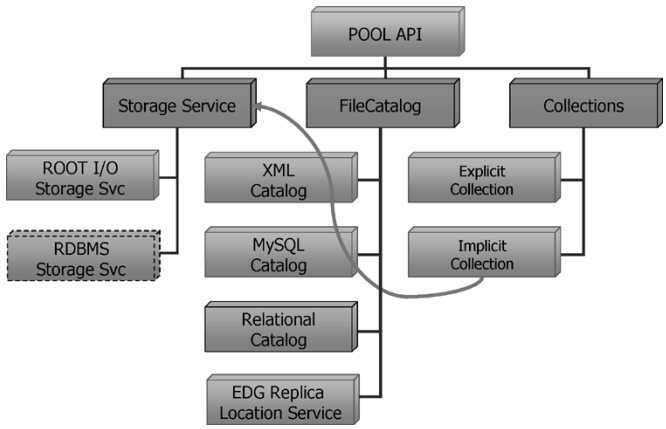

Fig. 1. Decomposition of the POOL API in three domains.

The task of the POOL framework is to store various types of data, such as event, detector, bookkeeping, and condition data. The data volumes associated with these types vary by many orders of magnitude, and the data are typically accessed in quite different ways according to their nature and role in processing activities.

The need to access widely varying data volumes with varying access patterns implies a need for a number of specific technologies to handle data persistency. For this reason, the POOL software architecture has been designed such that the best-adapted technology can be used transparently for each category of data. The POOL API has been defined to fulfill the common requirements of experiment data persistency, without exposing details of the specific backend technologies adopted. This feature means that the experiment software using POOL can be easily adapted to changing data handling technology over the LHC lifetime.

II. Pool Architecture

The POOL architecture has, therefore, been designed to provide a generic access to multiple persistency technology.

The POOL framework [2] has been developed in C++, the main programming language adopted by LHC experiments for the offline software. Part of the API is also available in python language through dedicated wrappers.

The API has been defined as three groups of public interfaces, representing the major domains supported: the Storage Manager, the File Catalogue, and object Collections (see Fig. 1).

The Storage Manager [3] domain is responsible for providing I/O for the data coming from several data sources: detectors, for example, or simulation and reconstruction programs. These categories of data are characterized by their large size and by the fact that they are written once, very often read back, and very

seldom updated. This access pattern is considered best served by C++ object streaming: therefore, the file-based ROOT I/O [4] was identified as a suitable choice for the first production implementation of the storage back end.

The File Catalogue [5] domain is responsible for maintaining consistent lists of data sets (files or databases) by mapping unique and immutable identifiers to strings which describe the physical locations (paths on file systems, connection strings for database replicas). These unique, immutable identifiers form part of the address of an object in the persistent storage. POOL provides three different implementations of the File Catalogue interfaces: XML, MySQL and EDG-RLS, serving the three main use cases of data handling on a network-disconnected computer, on a small processing unit and on a grid-connected system.

The Collections [5] domain supports the definition, creation, population, use, and management of ensembles of objects stored by means of POOL persistence services. These collections of objects have associated attributes which can be used for fast selection of the data during analysis procedures. Current implementations are based on MySQL tables and ROOT object structures. In addition, the same interfaces are exposed to the users for *implicit* Collections, defined as association by physical containment of objects in one or more specific databases.

The concrete implementations of the three domains' APIs follow a component-based modular structure. The components are assigned well-specified tasks, and interact via protocols defined using C++ abstract interfaces. This strategy allows the definition of common, low-level components for the handling to specific technologies, which can be used across the three domains.

III. Integration and Production Experience

The first two years of life of the POOL project have been spent in providing a functionally complete API, providing a ROOT-based implementation of the Storage Manager, and in developing the first implementations of File Catalogue and Collection interfaces.

Over the same period, great efforts have been made by each of the experiments to integrate the POOL software into their frameworks. Most of the POOL API has been integrated into the ATLAS, CMS and LHCb offline software and is regularly used in production activities, testing the scalability of real data taking processes in so-called *data challenges*.

The approach adopted by the three experiments in integrating POOL has been driven by the need to minimize the impact on already existing offline code, taking care in some cases to preserve the ability to read data already written with pre-POOL technologies. For this reason, for the object storage, the three experiments have integrated variations on the relevant POOL components, depending on their varying requirements for object navigation and object lifetime [6]. A more uniform approach has been adopted for the integration of the File Catalogue and Collections APIs.

The usage of POOL in three experiment data challenges, involving a total volume of ~400 TB of data, has been a first validation test for the whole architecture. In particular, the overall API has been proven to satisfy stability and reliability requirements. At the same time, some feedback has been collected in order to improve performance and extend the functionality and support of new technologies.

IV. Incremental Developments

The support of the data challenges during 2004 has required a change of focus from pure development to user support, deployment and maintenance. However, the stability demonstrated by the POOL software in large scale activities has allowed the POOL development program to continue relatively unhindered.

A. Migration to ROOT Version 4

The dependence of the POOL main object streaming technology on the ROOT framework requires constant adaptation of the POOL backend to the changes introduced in new ROOT releases. ROOT version 4, which has been recently published, has major changes affecting POOL, in particular I/O management. Among other advantages, ROOT 4 offers automatic schema evolution and a simplified streaming of standard C++ library containers. In order to profit from the new features of ROOT 4 and mark the changeover, POOL has started a version 2.0 development line.

The main challenge in this effort is to ensure backward compatibility for the data stored with the previous POOL version, which was based on the ROOT 3 series. This requires the automatic detection of the file format for C++ template-based containers, which is being resolved in a close collaboration between the ROOT and POOL teams.

The development process of the integration of ROOT 4 in the POOL 2 pre-releases is closely followed by the experiments, which verify compatibility with their previously written data.

B. File Catalogue Deployment

This year's data challenge productions have relied on XML and grid catalogue implementations. For the latter several weaknesses have been revealed over which POOL has little control. At the same time several new or enhanced catalogues are being developed. Moreover, changes in the computing models of the experiments are changing File Catalogue usage patterns, and need to be taken into account.

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

The new interface design is also intended to enable the synchronization of POOL File Catalogue interfaces with the API of the underlying grid services. A testing suite based purely on the POOL File Catalogue interfaces will be used by developers of new implementations to validate and benchmark their components.

C. Collection Catalogues

There are currently several implementations of components exposing POOL Collections interfaces. These are: Implicit Collections, implemented directly at the Storage Manager level and

Explicit Collections, implemented using ROOT trees or MySQL tables.

Cataloguing of Explicit Collections has been recently provided in response to experiment requests. Collection Catalogues are, in general, similar to File Catalogues; however, in Collection Catalogues the entries are named Collections instead of files. For the first implementation of the Collection Catalogues, we have reused the existing File Catalogue implementations and command-line tools.

Further development of 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.

V. A Relational Backend for Pool

A. Motivations and Goals

The first discussions of a relational back-end for POOL started in late 2003, triggered by interactions between the POOL team and the LHC experiments. It became apparent that there were two main physics use cases that had to be addressed, related to the LHC ConditionDB project [7] and the storage of configuration and detector data.

It had already become evident 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 configuration and detector control data are written by online processes 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.

B. 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 leads to the domain decomposition which is shown in Fig. 2.

The POOL relational back-end comprises three main domains.

- The Relational Abstraction Layer (RAL), which defines a technologically neutral API for accessing and manipulating data and schemas in relational databases.

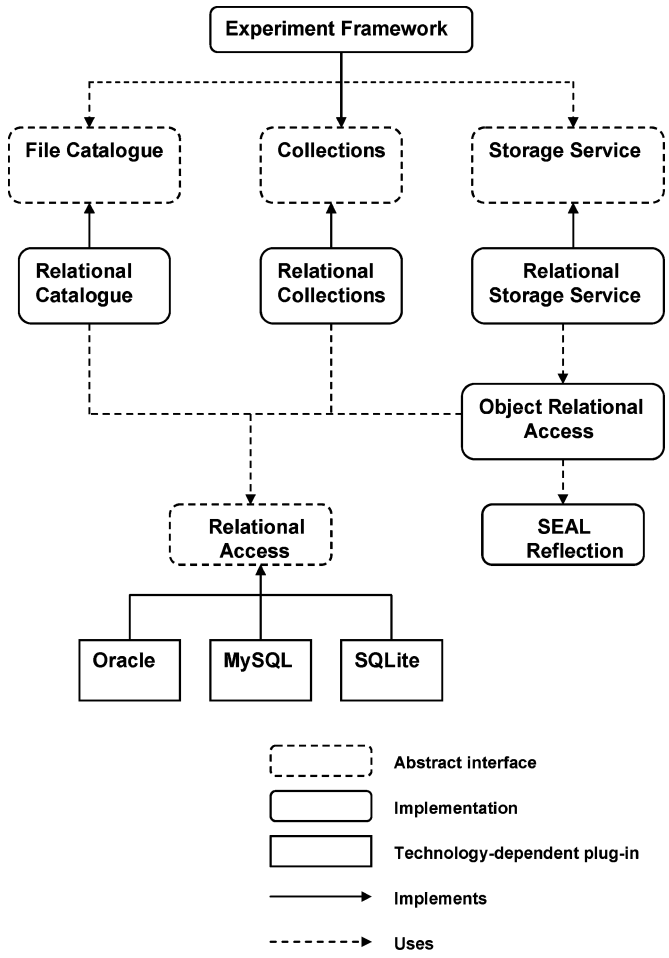

Fig. 2. The components comprising the POOL relational back-end and their relation to the rest of the software system.

- The Object-Relational Access 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 Access mapping mechanism.

C. 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; its utility in developing the relational components of POOL (File Catalogue, and Collections) and the ConditionsDB; and potentially its utility in enabling user application access to relational data. Moreover, its introduction may address the problem of distributing data in RDBMS' of different flavors.

The RAL abstract interfaces are defined in the *RelationalAccess* package. Their technology-specific realizations are implemented following the SEAL component model [8] as plug-in libraries. This architecture reduces the code maintenance effort for the relational components and allows efficient bug tracing. In providing access to specific RDBMS technologies via plug-ins, the risk of binding to a particular RDBMS vendor is minimized.

Furthermore it allows the usage of multiple technologies in parallel. Applications which access relational databases through the RAL automatically become testing grounds for plug-ins of new RDBMS flavors.

The RAL interfaces allow 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 description of the relational data is enabled 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 queries. A type converter implicitly manages C++ to SQL type conversion- and vice versa. Each technology implementation provides a default type mapping which is user customizable so that the user can take advantage of vendor-specific SQL type extensions.

The encapsulation of SQL types and syntax behind a C++ interface solves the problems which arise from the noncompliance of 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 to be loaded at run time is deduced from the technology field of a connection string 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 Catalogue. 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 for a 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 plug-in has been implemented using the Oracle Call Interface (OCI) client software. This choice was made for two reasons: the performance advantages that this solution offers; and the high probability of fewer configuration problems whenever POOL is released to compile with a new C++ compiler.

Since the first pre-releases of POOL 1.8 the Oracle plug-in has been 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, on encountering a 10g server, 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 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.

A plug-in which handles accesses to MySQL databases was also made available in the pre-releases of POOL 1.8. This library has been implemented using the ODBC API, meaning 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 reason was 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 changes considerably. The second reason was that the MyODBC driver exposes a more complete functionality, which allowed almost full implementation of the RAL interfaces. Finally, the third reason was that this plug-in could be 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 Catalogue. Some experiments have already integrated it in their frameworks and there are already experiment-specific applications accessing Oracle databases through the POOL RAL.

D. 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 is to be mapped to a relational structure, and vice-versa.

In the relational world tables are broadly equivalent to 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 derives from the associations 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.

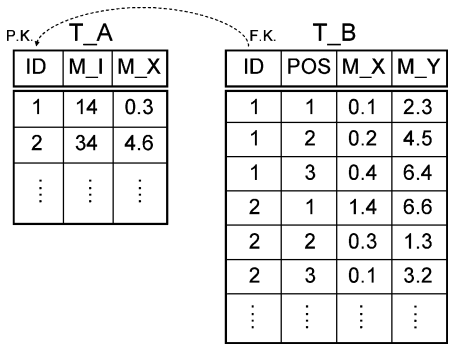

Fig. 3. Relational schema corresponding to a mapped class.

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 a simple C++ class which contains two simple members, and more complex vector member.

One of the possible mappings to a relational schema for this class is presented in Fig. 3. 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 software for generating mappings for a 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 generating mappings, a user can override them. This would be the case if, for example, one would like to generate object-relational mappings for existing data. POOL provides a tool which uses an XML file to steer mapping generation; the nondefault rules are specified using an XML schema. 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.

Object storage and retrieval is performed using SEAL reflection information for the C++ class of interest, 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 records 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 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.

VI. Summary

The LCG POOL project provides a software framework for the persistency of the LHC experiment data.

The main strategy of the project has been to satisfy the requirement to provide access to a variety of persistency technologies, allowing for possible changes during the LHC lifetime.

The software developed in the first phase of the project, integrates seamless a streaming technology (eg ROOT I/O) for complex object storage. During the last year the POOL persistency framework has been adopted by three LHC experiment (ATLAS, CMS and LHCb), integrated into their offline software and used in large-scale production activities. The POOL API has been fully validated and it has been demonstrated to meet most of the requirements for production.

More recently RDBMS technology has been introduced for consistent metadata handling with transactional access. In order to provide support for more relational database systems, the low-level code handling the hand-shake with the database has been factorized in a generic API (Relational Abstraction Layer) with a specific implementation for each DB vendor supported. In this way, all the database-specific SQL needed to operate on the different RDBMS is hidden by dedicated plug-ins, and all the POOL components access different RDBMS technologies through a single, uniform protocol.

A new relational back-end for the POOL Storage Manager has been also implemented based on the RAL, with the double purpose of storing data described as user-defined objects in relational table, or conversely presenting data already stored in relational table as objects. These functionalities are addressing two use cases not yet satisfied by the previous POOL components, such as the handling and management of Condition data and Detector online data.

The future POOL developments will be focused on tuning and improving the performance of data access, reducing the penalty introduced by the additional layers of the framework—RAL in particular—and optimizing the internal procedures for the data presentation as objects. Furthermore, the requests of the LHC experiments, necessitating changes in the existing code or implementation of new features, will be followed up in the POOL project plans.

References

[1] The POOL Project. [Online] http://pool.cern.ch
[2] D. Duellmann, "The LCG POOL project general overview and project structure," presented at the Proc. CHEP 2003, La Jolla, CA.
[3] M. Frank *et al.*, "The POOL data storage, cache and conversion mechanism," presented at the Proc. CHEP 2003, La Jolla, CA.
[4] R. Brun and F. Rademakers, "ROOT-an object oriented data analysis framework," *Nucl. Instrum. Methods Phys.Res. A*, vol. A389, pp. 81–86, 1997.
[5] Z. Xie *et al.*, "POOL file catalog, collection and meta data components," presented at the Proc. CHEP 2003, La Jolla, CA.
[6] G. Govi *et al.*, "POOL integration into three experiment software frameworks," presented at the Proc. CHEP 2004, Interlaken, Switzerland.
[7] A. Valassi *et al.*, "LCG condition database project overview," presented at the Proc. CHEP 2004, Interlaken, Switzerland.
[8] R. Chytracek *et al.*, "The SEAL component model," presented at the Proc. CHEP 2004, Interlaken, Switzerland.