ADDRESSING THE PERSISTENCY PATTERNS OF TIME EVOLVING DATA FOR ATLAS/LCG USING THE MYSQL BASED CONDITIONS DATABASE INTERFACE

A. Amorim*, Dinis Klose, Luis Pedro, Nuno Barros, Tiago Franco,
Universidade de Lisboa, CFNUL - Faculdade de Ciências , Lisboa, P-1749-016, Portugal
Andrea Valassi, Dirk Dullman, CERN, Geneva, Switzerland

Abstract

This paper describes the main developments introduced in the revised Conditions Database Interface deployed against the MySQL backend. The need for considering the different patterns of archiving and organizing the information along the time path is explained and the specific solutions introduced in the Conditions Interface to deal with the time interval and version description of conditions objects is discussed. The requirement of simultaneously sharing data among different applications in different environments, that share only the Conditions Database interface in common, motivated us to create a commonly used generic class for data that generalizes the relational table and that is independent of the object streaming solutions used. This not only allows the access to data that is not hidden by the serialization in some specific framework, but also enables the table projection and restriction on the database server for user data. The future directions of investigation to interface this generalized container to the user in different frameworks are discussed. The issue of distributed data storage and partitioning among long periods of time, is also addressed.

INTRODUCTION

Managing the calibration, alignment, configuration, control and monitoring of modern large hadron collider experiments like ATLAS presents remarkable challenges to the database systems. This document describes the main developments that were introduced in the MySQL version of the Conditions Database Interface to address some of these problems and discusses the new approaches that were required by the recent usage of the interface.

A common feature to the objects in the conditions database is the fact that they must be associated with a time interval of validity. They represent our knowledge of the detector for a given time period and must be associated, either through time or some other index, to all the events collected during that period. Although there must be a clear association between each event and the conditions that were used when processing it online, further improvement of the conditions for the same period leads to the concept of versions that must be packed in physically meaningful collections associated with tags.

* Antonio.Amorim@fisica.fc.ul.pt

The Conditions Database concept was first introduced in BaBar [3] as part of the support for information that was organized according to time intervals complementing the OODBMS system that was used for all data. CERN has picked up this effort [1] and made two implementations using first an object and later a relational database technology.

The development of the MySQL implementation that is described in this work [4] was motivated initially from the need to use online the Conditions Databases on platforms that were not supported by the vendor solutions used in the previous implementations. Since the MySQL implementation was relatively easy to deploy, both in the online and offline environments, we were able to concentrate our efforts on many important issues that were raised by its usage in different domains. These are both associated with the management of time intervals and versions, enabling any application to deal with the schema of the data that is stored and with the scaling of the system as the data gets accumulated in time.

Different specific solutions were introduced in the Conditions Interface to deal with the time interval and version description of conditions objects both for online and offline usage.

The development of a generic container that mapped a database extended relational table to a numerically optimized transient object, using STL containers, allowed us to incorporate the advantages of the relational features while directly mapping the transient objects to the tables in the database server. Very different applications, that share no common framework, can access the data in the database using this transient generic container.

The issue of distributed data storage and partitioning for long periods of time, is also important since it has implications on the design of the system and can only be accommodated by taking into account the different levels of indirection that are provided. These features provide an important handle on the scalability and load balance in a system that does not only aim to be completely distributed and achieve a very high performance for hundreds of simultaneous users, but also to fulfill the requirement that all the data collected during many years should be immediately accessible at all times.

THE EXTENDED CONDITIONS INTERFACE

This section describes the main extensions that were included in the Conditions Interface together with the main use cases that each of them addresses.

The generalized container

To allow data to be exchanged freely among different applications in an optimal way, we have mapped a generalized database table to a generic container that includes the schema and the data stored in or out of the databases. The framework specific objects that use an external serialization framework like ROOT or POOL are still supported trough the use of BLOBS.

The generalized container was defined through an abstract interface allowing different implementations optimized for different purposes. A single very optimized implementation, based on the use of variable type STL containers, was provided up to now. The generalized container is used both for data retrieval and data storage.

There are very interesting new developments that may have implications on the future implementation of this generic container, namely the relational implementation of the ROOT TTree class and the POOL relational interface. These avenues are being carefully investigated.

The different time patterns for data folders

The Conditions Database has a hierarchical structure which is divided into foldersets and folders. Foldersets can contain other foldersets and folders while folders can only contain data. In the initial implementation there was only one type of data objects which we shall call objects with versions. Each object is associated to an interval of validity and carries the data as a BLOB. When an object is inserted, the API checks if there is any overlap of existing objects. If there isn't, the object is simply inserted in the folder. On the other hand, if there is overlap, this means that there is a conflict of the intervals of validity which is solved by attributing a different version to the object that is being inserted. It is also possible to associate tags to sets of objects and then retrieve them by specifying the tag. In the initial implementation of the Conditions Database only folders which contained objects with versions were considered which allow the users to have different versions of a given calibration or alignment to apply to the detector. While this initial implementation worked well enough for certain types of data, such as calibration and alignment data, it was found to be inefficient for data with different characteristics, such as DCS data. Also, the fact that data is stored as a BLOB makes querying for specific values impossible. Top address these requirements, two new storage mechanisms were created, which take into account the different time evolution of online data such as DCS data:

- Tables,
- Tables with ID.

Tables This mechanism is used to store online data. When a value or set of values is inserted an interval of validity is defined such that the end time is equal to infinite, meaning that the interval is not closed. When a new value or set of values is inserted, the previous value is closed by setting the end time equal to the start time of the new value. In this way, the time evolution for a given value or set of values is continuous. The data itself is stored in a relational table, where each column represents one value and is of the same type as the value. This means that if you have a value which is of type integer, the column in the table will be of the same type. Supported types include integer, float, double, bool, string and array of each of these types. The fact that the data is stored in a relational table instead of a BLOB makes it possible to query for values (e.g. get all lines where value x$>$10).

Tables with ID This mechanism works much the same as the tables mechanism, the difference being that one can have different sets of data, which all have the same structure, in the same table. This means that you can have, for example, several chambers which have the same structure and properties and that you want to store their properties in the database. One way to do this would be to store each table in a different table. If, however, only the property of one of the chambers changes from time to time, and not of all chambers at the same time, it is more efficient to register only this change. In the case of tables with ID, an ID is associated to each chamber. When a property of one of the chambers changes, the respective values are inserted with the respective ID, and only the interval of validity of the previous entry with the same ID is updated. This way, each chamber is updated independent from the others.

The usefulness of being able to query for values led to the creation of yet another storage mechanism, called tables with versions. This mechanism is essentially the same as the original one, except for the fact that the data is stored in a relational table instead of a BLOB.

In order to make the API aware of the different storage mechanisms available and able to distinguish between them, folder types were created, associating a folder type to each storage mechanism. Since each storage mechanism represents a different time pattern, the folder types are associated to the different time patterns.

PARTITIONING IN TIME

The ATLAS experiment is supposed to last several years, during which a large amount of data is going to be collected. In order to improve performance, the data has to be partitioned over several servers. The best way to achieve this is by partitioning the data in time, dividing the data into blocks that correspond to large time intervals, e.g. three months. These blocks are then transferred to different servers, thus distributing the load of the central server over

several servers. Instead of handling all queries, the central server redirects to the server of the respective time interval. All of this is done at low level so that the user doesn't have to know about any of it. The user just makes the request and the API takes care of finding the right server to perform the necessary action, using a partition table to check which server has the requested data. Setting up the partition is an administrative task, meaning that the DB administrator has to perform the necessary actions to create the partitions and moving them to the right server. It is possible to write tools based on the API to perform part of the work.

INTERFACES IN ATLAS

For the widespread use of the Conditions Database in the ATLAS experiment it was necessary to build a set of interfaces. There are multiple environments in which the users would like to store data in the Conditions DB or access the data stored in it, which led to the development of a set of interfaces that make this possible for a wide range of environments.

The ATHENA framework interface

The Athena framework is the common framework in which the physics analysis will be made. This framework is developed in C++ and is a component software. It was a vital requirement that the users would be able to access the data from this interface to make analysis. For this purpose a conversion service was developed. This conversion service is responsible for the conversion of the transient generalized container in the Conditions Database to the generalized container in the Athena framework, the GenericDbTable. This generic transient object is nothing more than a implementation of the generalized container for the Athena framework, having the same functionalities. To read data into the Athena framework, the user must send a request to the DetectorStore, which will call the service responsible for the interval of validity of the object (IOVSvc) and the service that should get the data itself. The latter functionality is the responsibility of the CondDBMySQLCnvSvc. For the moment, there is also a simple facility to store data into the Conditions Database, but further implementations are being analyzed.

The PVSS Interface

The PVSS API Manager is, as the name says, an API which links PVSS to the Conditions DB. Basically, the user defines a list of datapoint names in a specific datapoint container (see below) and sets up the database configuration. The manager then connects to the datapoints specified by the names creating entries, for each one, in the database (each datapoint is one folder in the Conditions Database). The initial values of these datapoints are stored, and any change of value, is stored in the Conditions Database through the storage mechanism for online data (tables and tables with Id) described above. The main features of this manager are:

Structure handling The manager is able to store structures as such, that is, it creates a table which reflects the structure of a datapoint. This includes the possibility of any of the entries being a dynamic array.

Exception handling The manager uses exception handling, making fault diagnostic easier.

Datapoint container To store datapoints, their names are stored in a datapoint. Changes to this datapoint can be made on-the-fly, meaning that the manager reacts automatically to any change in the list of the datapoint names.

Configuration The configuration of the manager, that is, setting the values for database connection, is done through a specific datapoint which also contains status elements which hold the running status and the storage status (running/not running). It is possible to make changes on the fly, without having to stop and restart the manager. In order to facilitate the configuration process, a graphic interface panel is provided which makes it easy to set the values of the configuration datapoint.

The Online Conditions Database Interface

In order to make Conditions data coming from the Online Software available, it was necessary to create an interface between the Online Software and the Conditions Database. This interface, which is called the Conditions Database Interface (CDI), is responsible for the collection of data grouped by the information services (IS) and the storage of the collected data in the Conditions Database in a structured way. The IS data is structured into tables where entries change value over time. The data coming from IS is stored in the Conditions Database via the online storage mechanism of the extended API and using the generalized container preserving the structure.

The Web Browser

A lot of users have stated the need for a tool of easy access which would allow to look at the data in a way similar to the way it is classified, by a hierarchical structure (folders) and a time of validity. For this purpose a tool was developed that could be used in any environment and would provide a simple access to the data by the users. This tool was written in PHP and uses an identification mechanism similar to the Conditions Database to grant access for the users. This tool allows the users to see the data stored using the generalized container in a way very similar to the structure of the data in the container. For objects stored as blobs (eg. NOVA objects) the browser provides a link to the location where the data is in fact located.

USAGE IN THE TESTBEAM

The current implementation of the Conditions Database has been successfully used in the 2004 ATLAS Combined Testbeam to store conditions data. Specially the extensions for online storage have been heavily used to store data coming from the DCS and the Online Software.

FUTURE DEVELOPMENTS

For future developments several aspects are being looked into:

Implementations

Currently there is one implementation in Oracle of the original interface and one implementation in MySQL of the extended API. Current planning foresees making an implementation in Oracle of the extensions and reviewing the MySQL implementation. The possible use of the POOL RAL is also being investigated.

Interface

The current interface, the original as well as the extensions, is not very intuitive in its usage. It is planned to redesign the interface in order to make it easier to use by reducing the methods needed and the number of classes. The methods themselves are also going to be simplified.

Extensions

The current generalized container is already very versatile. However, it's interface is not very intuitive adn there are limitations concerning the possible datatypes. Future plans include making the interface easier to use and extend the generalized container in order to accommodate generic objects as column type and the possibility of having columns with variable type. Currently there are four different storage mechanisms (objects with versions, tables with versions, tables and tables with Id). Additional storage mechanisms are being investigated, more specifically a hybrid mechanism which combines the online storage and offline storage mechanisms to make it possible to store and edit online type data.

Tagging

The most important aspect however, is the tagging mechanism. Until now, it was only possible to tag the HEAD. This, however, is not enough to satisfy the user requirements.In order to make tagging more general and versatile several new tagging options are being considered:

- Tag from tag: create a tag with the objects of another tag
- Tag to tag: hierarchical tags that point to several other tags
- Create tag or Re-tag to the old head with insertion time less or equal to a given time (from BaBar)
- Use a re-tag time interval that only changes objects that are contained in a user time interval

These tagging options should be general enough to allow the users to tag any object or set of objects they want.

CONCLUSIONS

From all the challenges involved in the addressing of the Conditions Database system, there were two that deserve a special consideration. The first regards the heterogeneity of the platforms involved, and the solution adopted was the development of a common API that gives access transparency[5]. The advantages of this approach are visible when the several tools supported by Conditions work independently of the DBMS that is used. The second challenge regards the problematic of dealing with time varying data under the available DBMSs. Our approach is based on a temporal extension to the relational model, where the implementation of time based primitives gives the user/programmer a high level of abstraction to manipulate execute DML and DDL on time based relations[7].

ACKNOWLEDGEMENTS

This work was supported by *Fundação da Ciência e Tecnologia* under the grant POCTI/FNU/43719/2002.

REFERENCES

[1] Stefano Paoli, Conditions DB Interface Specification, CERN-IT Division

[2] Andrea Valassi et. al., , CHEP 2004 Conference, Interlaken, Swiss, September 2004

[3] Igor Gaponenko, An overview of the BaBar Conditions Database, CHEP2000 Conference, Padova Italy, January 2000.

[4] A. Amorim, J. Lima, L. Pedro, D. Klose, C. Oliveira, N. Barros. IEEE-NPSS: An Implementation for the ATLAS Conditions Data Management Based on Relational DBMSs. In Proceedings of the 13th IEEE-NPSS Real Time Conference, pp. 591- 595, May 2003.

[5] G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems - Concepts and Design". Addison-Wesley, 3th Edition, 2000

[6] M. Bhlen et al. TDB Glossary. Available: http://www.cs.auc.dk/ csj/Glossary/

[7] Abraham Silberschatz, Henry F. Korth and S. Sudarshan. "Database System Concepts". McGraw Hill, 4th Edition, 2001