LHCb Conditions Database

M. Clemencic, J. Palacios, CERN, Geneva, Switzerland
N. Gilardi, University of Edinburgh, Edinburgh, UK

Abstract

The LHCb Conditions Database (CondDB) project aims to provide the necessary tools to handle non-event time-varying data. The LCG project COOL provides a generic API to handle this type of data and an interface to it has been integrated into the LHCb framework Gaudi. The interface is based on the Persistency Service infrastructure of Gaudi, allowing the user to load it at run-time only if needed.

Since condition data are varying with time, as the events are processed, condition objects in memory must be kept synchronized to the values in the database for the current event time. A specialized service has been developed independently of the COOL API interface to provide an automated and optimized update of the condition objects in memory.

The High Level Trigger of LHCb is a specialized version of an LHCb reconstruction/analysis program and as such it will need conditions, like alignments and calibrations, from the conditions database. For performance reasons, the HLT processes running on the Event Filter Farm cannot access the database directly. A special Online implementation of the CondDB service is thus needed under supervision of the LHCb Control system.

INTRODUCTION

In an experiment it is fundamental to handle informations that are related to the status of the detector at the time of the acquired event. This type of time-varying non-event data are often grouped under the term "conditions".

Each condition datum is identified by a category (the context to which the datum is meaningful, ex.: "temperature of a gas enclosure") and the period in time in which the datum is valid (*Interval Of Validity* or IOV). In many cases it is needed to be able to replace the value of a condition with a new one (ex.: new alignment constants can be measured with improved algorithms), still being able to use the old value if needed. The condition data that can exist only in one version, like the numbers obtained by reading a probe (temperatures, pressures), are called "single-version", while condition data that may exist in many versions, like the numbers that are obtained off-line calculations (alignment, calibrations), are called "multi-version" and need a third parameter to be identified uniquely: the "version" (Fig. 1).

LHCb needs an infrastructure to handle condition data

from a conditions database which is:

- integrated in the framework (Gaudi),
- flexible enough to be able to handle as many use cases as possible,
- efficient, avoiding as much as possible useless operations
- easy to use.

Persistency of condition data will be achieved through the library COOL which is being developed by the LCG (LHC Computing Grid) group in collaboration with LHCb and Atlas. COOL has been designed to handle time-varying non-event data in a generic way, using a Relational Data Base Management System for the storage[3].

LHCB CONDITIONS DATABASE

The conversion between the persistent representation of the condition data and the transient one is obtained extending the Gaudi Persistency Framework.

Gaudi Persistency Framework

The persistency framework of Gaudi is based on "data services" each handling a "transient store" which is populated via requests to a "persistency service"[1]. The persistency service is a dispatcher that forwards the requests to the most appropriate "conversion service" that, through its pool of converters, is able to perform the actual conversion from the persistent format to the transient one (Fig. 2).

LHCb is using a few data services, each of them specialized in a particular type of data. Between them, we have the "event data service" that holds event data and is cleaned every event and the "detector data service" that gives access to the objects related to the detector description and it is not cleaned until the end of the job[2].

Since objects containing condition data are mainly used by detector description objects and their lifetime is independent from the event loop (a condition is usually valid for more than one event), we choose to extend the persistency service of the detector data service for the conditions database.

Another reason to use the detector data service for conditions, is that the format used for persistency is XML, which is a good compromise between human-readable and machine-readable.
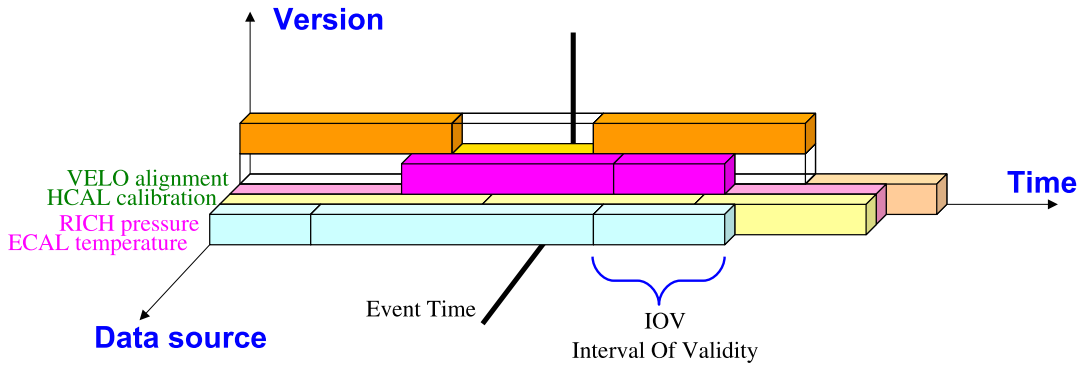
Figure 1: Graphical representation of the space of condition data. "RICH pressure" and "ECAL temperature" are single-version conditions while "VELO alignment" and "HCAL calibration" are multi-version conditions.
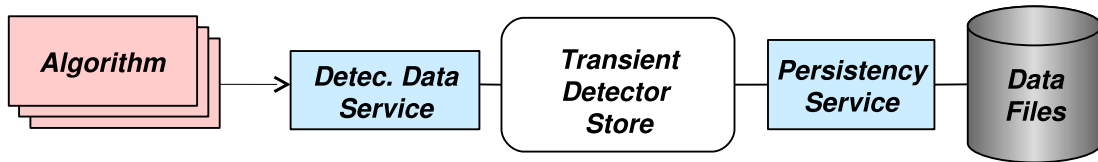

Figure 2: Schema of the access to persistent data in Gaudi.

Condition Database Conversion Service

The "CondDB conversion service" is a specialized conversion service with its own converters and a pool of "CondDB access services" that work as interfaces to the COOL API.

Hiding the COOL API behind an "access service" allow to absorb changes in the API (which is still in fast development) and implement some high level functionality that are not available in COOL.

Currently we expose, through the CondDB access service, only a simplified version of COOL API avoiding the details that are not needed for our purposes. At the same time we implemented a cache to reduce the number of round trips to the database and to allow to have an in-memory CondDB.

We also provide the possibility to use more than one CondDB. If a condition datum is not found in the first database, the following one is used. This allows developers to work on a small local CondDB instance containing only the condition data they are interested in and retrieve the others, needed by other parts of the application, from the master CondDB.

First Tests

The content of the LHCb CondDB instance is not yet finalized, thus we tested the infrastructure using the content of our detector description XML files.

Compared with the file-based detector description, the CondDB-based one is 4 to 8 times slower. We already identified the main sources of this poor performance and we are working to solve them.

UPDATE MANAGER SERVICE

During a reconstruction or analysis job, the transient representations of the condition data have to be kept synchronized with the values valid for the event being processed. We have also to consider that the condition data may not be useful as they are, but need to be digested or preprocessed in some way, so we need a system of call-back functions to call if a given condition datum has been updated in the transient store. All those task are performed by a dedicated service: the "update manager service".

Network of Dependencies

The update manager service is keeping track of a network of dependencies (Fig. 3). Any object can be registered to the service as consumer for another object, providing at the same time a pointer to a member function to call when the used object is updated.

The requirements on the consumer object are minimal: the member function to call must comply to a predefined signature. Each consumer object may implement more than one call-back function and each call-back function may need more than one condition object.

The object used by the consumer is even less restricted, it can be anything, e.g. another consumer object.

Update Policy

When an update is needed, the network of dependencies is navigated top-down, from objects which are not used by any other object (head objects), to objects that do not use any other object (leaves). When a condition object needs to be updated, the new values, retrieved from the CondDB,
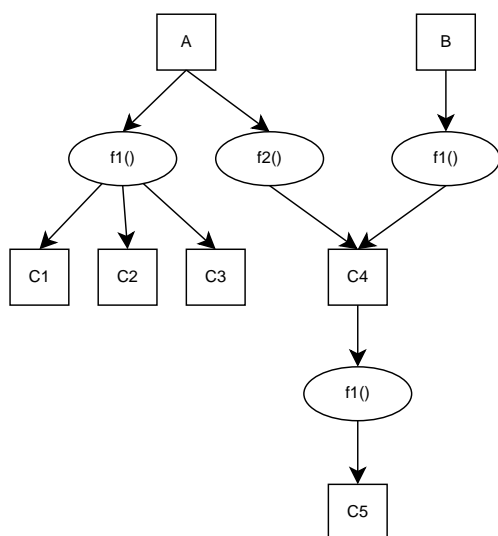
Figure 3: Example of a network of dependencies handled by the update manager service. Objects `A` and `B` are consumer objects, `f1()` and `f2()` are member functions. `C1`, `C2`, `C3` and `C5` are simple condition objects while `C4` is a complex condition object which is also a consumer of a condition object.

replace the old ones. Once that all the condition objects used by a given call-back function have been updated, the function is called. When all the call-back functions of a given consumer object have been called, the control goes up one level to the consumer of the current object, if any.

Considering the example of Fig. 3, when the datum of the condition object `C5` is replaced by the new one, we need to call `C4->f1()`, then `A->f2()` and `B->f1()`.

In order to reduce the number of redundant checks, we compute the intersection of the IOVs of the objects used by each registered call-back function and we use it as overall IOV for the function. In the same way we use as IOV of a consumer object the intersection of the IOVs of all the member functions registered for the object. The cached overall IOVs permit to know if any of the call-back functions of a consumer object need to be called, without following the dependencies. We also cache the intersection of IOVs of all the head objects so, when there is no need of an update, we check only one IOV.

ON-LINE

During data taking periods, we will have ~ 4000 concurrent processes running in the pit, that need to access condition data, mainly at initialization time. A database server will not be able to handle this load, so we plan to publish condition data to the on-line processes in a different way.

When the on-line processes have to be initialized, a dedicated process will read the data from the condition database and will send them to a service in each on-line process that will fill the cache of the CondDB access

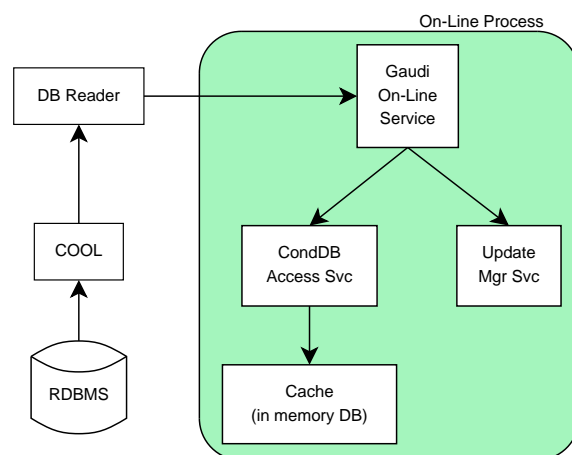service and notify the update manager service about the values inserted (Fig. 4).

Figure 4: Schema of the information flow from the RDBMS to an on-line process.

Condition data produced in the pit (like readings of probes), will be sent to a dedicated process that will write them to the CondDB. It may also happen that some of this condition data need to be published to the on-line processes while they are running. If it is the case, the DB writer process will also send the data to the on-line process like it happens during the initialization phase.
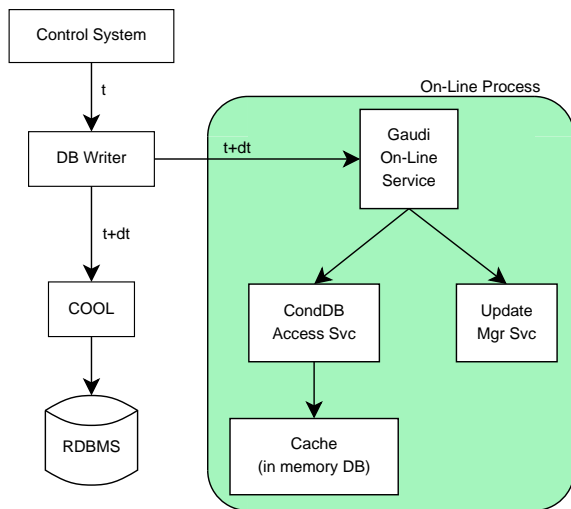
Of course, the propagation of the data from the control system to the running processes is not instantaneous and it may happen that when a condition valid from time t_0 reaches the on-line processes, an event with time $t_1 > t_0$ has already been processed. In such a situation, the reproducibility of the results of the trigger process is compromised, because there is no way of knowing, a posteriori, which condition value has been used for that event. To avoid the problem, the initial time of the condition datum will be propagated to the CondDB and the on-line processes increased by a Δt that takes into account the latency of the system (Fig. 5).

DEPLOYMENT

The LHCb conditions database will be accessed:

- at the pit, for reading (by the trigger processes) and for writing (by the control system)

- at CERN, for reading (reconstruction and analysis processes) and for writing (by CondDB administrators)

- on the GRID, only for reading

The deployment plan is to have the master copy of the conditions database located at CERN and, for load balancing and low network latency, a read-only copy in each Tier-1 center, all of them based on Oracle®. The replication process will be realized via "Oracle streams".

Figure 5: Schema of the information flow from the control system to the CondDB and to an on-line process.

The pit software system must not depend on the connection to CERN, so it is mandatory to have a copy of the database located at the pit too. The copy at the pit will be accessible both for reading and for writing. To avoid possible conflicts between condition data written in the pit with those written in the master copy at CERN, we will clearly separate the database in two sections: an "on-line" one and an "off-line" one. A schematized view of the deployment plan is show in Fig. 6.
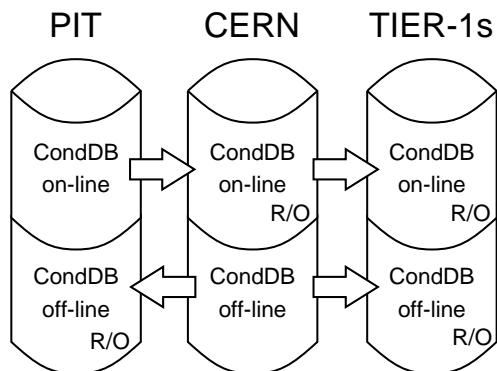

Figure 6: Schema of deployment plan. The arrows represent the direction of the replication via Oracle® streams.

REFERENCES

[1] M. Cattaneo, "GAUDI - The Software Architecture and Framework for building LHCb Data Processing Applications", CHEP2000, Padova, 2000.

[2] R. Chytracek, "The LHCb Detector Description Framework", CHEP2000, Padova, 2000.

[3] A. Valassi, "COOL Development and Deployment - Status and Plans", CHEP06, Mumbai, 2006.

[4] [LHCb Collaboration], "LHCb TDR computing technical design report," CERN-LHCC-2005-019.

[5] LCG COOL Project web site, http://lcgapp.cern.ch/project/CondDB