Journal of Physics: Conference Series **664** (2015) 042024 doi:10.1088/1742-6596/664/4/042024

The CMS Condition Database System

S. Di Guida[1], G. Govi[2], M. Ojeda[3], A. Pfeiffer[3], R.Sipos[4]
on behalf of CMS collaboration
[1] Università degli Studi Guglielmo Marconi, Roma, Italy
[2] Fermi National Accelerator Laboratory, Batavia ILL, USA
[3] CERN, Geneva, Switzerland
[4]Eotvos Lorand University, Budapest, Hungary

E-Mail: giacomo.govi@cern.ch

Abstract. The Condition Database plays a key role in the CMS computing infrastructure. The complexity of the detector and the variety of the sub-systems involved are setting tight requirements for handling the Conditions. In the last two years the collaboration has put a substantial effort in the re-design of the Condition Database system, with the aim at improving the scalability and the operability for the data taking starting in 2015. The re-design has focused on simplifying the architecture, using the lessons learned during the operation of the Run I data-taking period (2009-2013). In the new system the relational features of the database schema are mainly exploited to handle the metadata (Tag and Interval of Validity), allowing for a limited and controlled set of queries. The bulk condition data (Payloads) are stored as unstructured binary data, allowing the storage in a single table with a common layout for all of the condition data types. In this paper, we describe the full architecture of the system, including the services implemented for uploading payloads and the tools for browsing the database. Furthermore, the implementation choices for the core software will be discussed.

1. Introduction

The "Conditions" are non-event data resulting from calibration and alignment algorithms. They vary with time and they need to be accessible by physicist within the offline event-processing applications. The CMS computing infrastructure has a dedicated software system managing such data, called Condition Database.

The successful CMS Run I data taking (2009-2013) has lead to the discovery of the Higgs Boson in 2012. During this period, the Condition Database has been used at a very large scale. Thanks to the experience gained during this phase, the underlying requirements of the system were analyzed in depth and are now much better understood. In particular, the significant operational effort that the infrastructure required has motivated a re-design towards the Run II phase. The goal was to provide the experiment with a lighter and therefore more operational structure.


Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

2. Analysis of past experience

In terms of scalability and maintenance, all of the issues identified belong to two main categories:

- *Complexity of the software*. The system running at Run1 was based on a Object Relational Mapping (ORM) approach, allowing the mapping of arbitrary C++ objects to relational entities (Tables, Columns, Indexes)[1]. This approach required the use of a Root-based package for the C++ class introspection. The knowledge of the "internals" of the condition classes was essential for establishing an automatic generation of the mapping to the relational entities. This choice implied a significant number of software layers on top of the database access layer.
- *Complexity of the infrastructure*. As a drawback, the ORM approach generates a large number of relational tables (an average number of 3-4 tables per C++ class). To cope with possible clashes in the naming of relational entities, several schemata were used to store the experiment conditions, assigning one or more dedicated schema to every subsystem. This choice allowed improving the scalability of the storage, but it also increased the operational effort required to administrate the infrastructure.

3. The Condition Database System for Run II

The new Conditions Database system has been designed and developed over the past two years. Here we briefly discuss the abstraction process which has been followed for the design and implementation, starting from the analysis of the use cases, the description of the data model and the base technology choices.

3.1. Data Model

Conditions data change with time and require frequent lookups by time in the data store.
The bulk data is the ***Payload***, representing the set of sensible parameters consumed in the various workflows of the physics data processing. The *Payload* is associated to an user-defined C++ class (*Payload Type*), containing information related to a wide range of subsystems, including detectors, triggers, beam monitoring and physics algorithms. The Payload data is exchanged and stored as an unstructured binary array, with no assumption on its internal layout.
In the CMS software framework, a special C++ class (the ***Record)*** acts as entry point for a Payload. Every processing workflow for event data involves a specific set of Records, each of them requiring valid conditions.
The time information for the validity of the Payloads is specified with a parameter called *Interval Of Validity (**IOV**)*. In this context, time is represented by a Run number (identifier of the event data set collected in a given time span), luminosity section id or an universal timestamp.
A fully qualified set of conditions consists of a set of Payloads and their associate IOVs covering the time span required by the workload. A label called ***Tag*** identifies the version of the set (Fig.1). In the CMS model, the IOV sequence identified by a Tag is always consecutive with no holes or overlap in time. This choice allows defining the interval by only the lower bound (the "*since*"), the upper bound being set by the "since" of the following IOV in the sequence**.** The last available IOV in the sequence has by default an infinite upper bound (open IOV). In this way, the validity of a given Payload does not need to be continuously extended. This approach ensures that the update requests, requiring the access to the database, are limited to the changes involving new payloads.
A collective label called ***Global Tag*** identifies the set of Tags assigned to the Records involved in a given workflow. In this way, the production managers can handle specific collections of Tags and configure various jobs to access different subsets of them for large-scale data production.

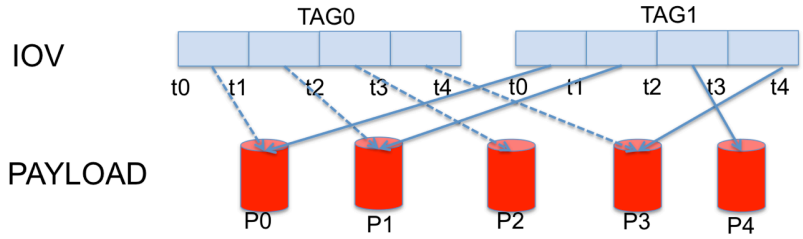

Figure 1: The IOV sequence for a given Payload set, identified by a Tag.

3.2. Implementation choices

The data model described in the previous section has been designed to fulfil the requirements imposed by the CMS software framework. The object-oriented nature of the offline software framework requires the Condition Data to be delivered as objects. Given the heterogeneous nature of the systems providing Condition Data, there is no pre-defined layout for the Conditions. The various systems are free to describe them with arbitrary C++ classes. The Payload objects, instances of these classes, are streamed into binary arrays and stored as Binary Large Objects (*BLOBs)*. This storage model allows hiding the details of the several Payload Class implementations, removing a considerable part of the complexity from the system.

The conversion of the Payload object in memory from the C++ Class layout to the binary array layout as stored in Conditions is called Serialization. This process is non-trivial, since it requires some level of intrusion into the class to serialize, or alternatively class introspection capability to provide knowledge about the data members at run time. Among the various option considered, we have adopted the serialization package provided by the *boost* libraries. This choice is mainly motivated by:

1. The portability across platforms.
2. The stability and reliability of the package.
3. The support of multi-threading.

Details about 1 and 2, and the performance study related to 3 are reported in [2]. The boost serialization mechanism is basically intrusive for the class to serialize, since it requires either public access (i.e. setter/getter methods) to the data attributes, or the implementation of the (private) serialization/de-serialization functions. To minimize the impact on the large sets of payload classes to serialize, a dedicated build infrastructure has been set up to generate automatically the 'intrusive' code required by the boost library. This solution has been implemented with a python tool, using the C++ analyzer provided by the *clang* library. This tool iterates over the various classes defined in their header files, identifying data members and the associated types, and generating the required serialization code.

As previously described, in a processing workflow, the condition data are specified by a Global Tag. The retrieval of the Payload, containing the sensible parameters, is based on an iterative selection process:

1. Select the list of Tags assigned to the required Records in the specified Global Tag.
2. Select the list of IOV sequences and their corresponding Payloads.
3. Select the specific Payload valid for the required "timestamp".

This selection process, involving strings or numbers, can be naturally implemented with a Relational Database Management System (RDBMS) technology, where these quantities can be easily indexed. In addition, RDBMS are suitable for handling concurrent random access to a centralized storage. The confirmation of Oracle as the main storage technology (already the case for the Run I) has driven the choice. The motivations for this choice are:

- Scalability. The requirement for storage of the order of several Terabytes is well supported and rather common in the applications in the industry.
- Reliability and resilience. Oracle is a well-known technology, requiring expertise relatively common in the job market. The wide community of users guaranties that the information is spread and bugs are usually known and fixed regularly.
- Database Administrator (DBA) service and support is provided at CERN by the IT department on a 24x7 schedule. This is a key point, since it accounts for a large operational effort: setting up and maintenance of the servers, management of the software, backups, and support for query performance.

In addition to that Oracle based central storage, the users can store their data in private files, which are also serving as interchange format. These files are handled with the SQLite software, using the same database structure as for the master database. The interoperability between Oracle and SQLite storage is achieved with the CORAL software. CORAL [3] is a package developed by CERN IT, providing a C++ abstraction layer for relational database. It allows implementing SQL free code for data selection, insertion and update on relational databases.

3.3. Database model

As mentioned previously, one of the aims of the system re-design has been to reduce the complexity of the database schema. In fact, the data model presented in the previous section fits in a relational database in a schema with only 5 tables, as shown in Fig. 2.

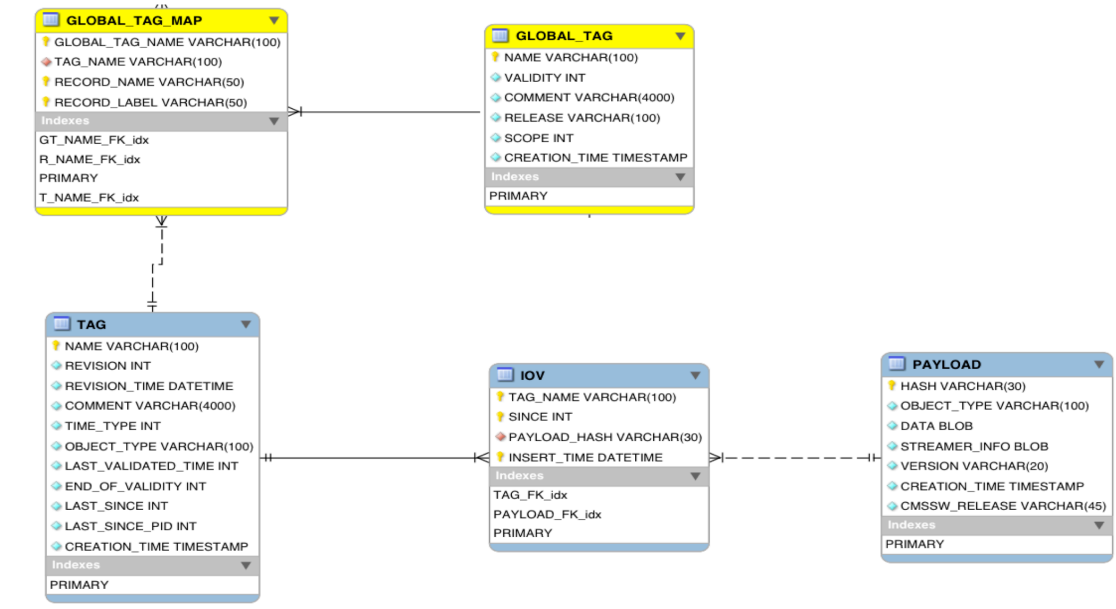

Figure 2: The Relational Schema for the Condition Data

The benefits of this design are:

- The queries are simple, involving indexed strings for label-based metadata (for TAG and GLOBAL_TAG).

- The time search for the IOV, which could be expensive, can be partially moved to the client side. A special index can be built dynamically and cached on the client side, to allow retrieving the IOV data in 'pages'.
- Payloads can be uniquely identified by their 'hash' value, calculated by combining the class name and the binary array serialized data.

3.4. Applications.
The use cases identified in the experiment operation, have been addressed by specific implementations:

1. Consuming the Conditions required by specific workflows:
This case includes the High Level Trigger (HLT), the Bulk Data processing and all the reprocessing campaigns. These workloads are submitted over multiple jobs in computing farms, all of them requiring to query the database. To support a large number of simultaneous connections, and limit the load to the database server, a middle tier running Frontier [4] has been added in front of the Oracle Server. Frontier stores the results of the queries in a hierarchical set of Squid proxies, providing an efficient load-balanced caching system.
2. Updating the Conditions for a given subsystem:
The process of refining calibrations is performed by the various subsystems with 'private' processes, which are generally asynchronous. To provide a central access point for the upload of the new data set into the database, a special service has been deployed, the Condition Uploader. The aims of this service are:
    - Solve the security-related issues for the access to the master database, sitting inside the firewall in the CMS online computing environment.
    - Centralize the authentication and authorization for the update access.
    - Synchronize – when required – the run-based time validity with the ongoing processing state of the target workflow.
    - Insert the data uploaded into the Oracle database.
    - Log the upload requests.

3. Tools
The Conditions involved in a typical production workflow are usually up to a few hundreds records (Tags). This implies the existence of a large number of Tags and Global Tags in the database. To allow inspection of the available items and to make searches, we have created a web-based Browser application. The Browser is able to search a user keyword in all the metadata columns of the 5 tables. The result of the search is presented in a categorized way per Tag, Global Tag and Payloads. It is also possible to list the content of the Tag, Global Tag and Payloads selected in query result.
Similar features have been provided in a set of command line tools based on python, which allows to search, list, edit and copy Tags and Global Tags.
Since the tools for the data management are independent of the event processing, Python was chosen as the base language because of its simplicity and flexibility. All the global tag management software are written directly in Python while at lower level, database connection and transaction management and query abstractions are exposed to Python via the *sqlalchemy* package, a standard binding layer for RDBMS in Python.

4. Conclusions

CMS has implemented a new infrastructure for the management of the condition data for Run II. The design has taken advantage of the experience and the knowledge gained during the data taking in Run I. The data model has been simplified into only a few entities: Payload, Record, IOV, Tag and Global Tag. The C++ payload Objects are serialized into binary arrays using the boost serialization library and a python script to automatically generate the code for the serialization of the payload classes. In this way, the Payload can be handled as an unstructured data entity. This choice has allowed to significantly reduce the complexity of the software components. Also the database infrastructure has been simplified. Storing the Payload as BLOB type, the data model proposed fits into a relational schema with only 5 tables. A set of tools has been provided to the production managers for the tag and global tag management and data transfer. Services such as the web service and the conditions browser facilitate interaction with data by physicists.

References

[1] G.Govi *et al*. 2010, *CMS Offline Conditions Framework and Services,* Proc. Conf. for Computing in High Energy and Nuclear Physics
[2] A.Pfeiffer *et al*. 2015, *Multi-threaded Object Streaming*, Proc. Conf. for Computing in High Energy and Nuclear Physics
[3] I. Papadoupolos *et al*. 2006, *CORAL a software system for vendor-neutral access to relational databases*, Proc. Conf. for Computing in High Energy and Nuclear Physics (Mumbai)
[4] Dave Dykstra. 2010, *Scaling HEP to Web Size with RESTful Protocols: The Frontier Example,* Proc. Conf. for Computing in High Energy and Nuclear Physics