THE DEVELOPMENT OF A NEW CONDITIONS DATABASE PROTOTYPE FOR ATLAS RUN3 WITHIN THE CREST PROJECT

E.Alexandrov[1], A.Formica[2], M.Mineev[1,a], S.Roe[3]
on behalf of the Software and Computing Activity

[1]*Joint Institute for Nuclear Research, Joliot-Curie 6, 141980 Dubna, Moscow region, Russia,*

[2]*Université Paris-Saclay, CEA/Saclay IRFU, 91191 Gif-sur-Yvette, IRFU/CEA, France,*

[3]*CERN, CH - 1211 Geneva 23, Switzerland*

E-mail: [a] mineev@jinr.ru

The CREST project for a new conditions database prototype for Run3 (intended to be used for production in Run4) is focused on improvement of Athena based access, metadata management and, in particular, global tag management. The project addresses evolution of the data storage design and conditions data access optimization, enhancing the caching capabilities of the system in the context of physics data processing inside the ATLAS distributed computing infrastructure. The CREST architecture is designed as a client server model, with the storage backend implemented in a relational database. The data access was realized with a pure REST API with JSON support. The new C++ client access library provides an HTTP query interface. A tool to convert the existing conditions data (stored in Oracle and accessible via the COOL API) into the new CREST system using a custom JSON format has also been implemented. Preliminary data migration has been done to allow testing data retrieval from Athena and the process of validation of the server and client functionalities is in progress.

Keywords: ATLAS, conditions database, CREST, REST, JSON

Evgeny Alexandrov, Andrea Formica, Mikhail Mineev, Shaun Roe

1.Introduction

This article presents a description of the CREST project [1]. This project is a new database prototype for Conditions data with REST interface (CREST) for the ATLAS detector [2] at the Large Hadron Collider (LHC) at CERN. Conditions data are non-event data, used to describe the detectors status, and constitute an essential ingredient for the processing of physics data, in order to reconstruct events optimally and exploit the full detector's potential.

Conditions data consist of time varying quantities like detector calibration and alignment data, electrical and environmental measurements (such as voltages, currents, pressures), temperatures, run and data acquisition configuration information, LHC beam information, trigger configuration and detector status data.

During LHC Run-1 and Run-2, ATLAS Conditions data were managed by the COOL/CORAL system (by CERN-IT). COOL is a C++ API based on CORAL client for access to the Relational (Oracle) DB [3]. This COOL based DB realization has its own high level functionalities. All detector sub-systems have their own “COOL databases” (*Schemas*) and store payload data in dedicated tables (*Folders*). Payload data are stored according to Interval Of Validity (IOV). Each IOV is defined by a since time and an until time. COOL has two Folder types. The first one is single-versioned Folders: IOVs can be appended here (no overwrite, no Tag defined). The second one is multi-versioned Folders: IOVs can be stored in arbitrary way and are grouped in Tags (thus allowing the payload to be overwritten). A Global Tag is a collection of multiple Folder Tags. Payloads are retrieved by providing IOV boundaries (and related Tag name).

1.1 Motivation for a new database

The new condition database project was started for several reasons. COOL DB caching is not well optimized since queries defined using different IOV boundaries may return the same payload data: this affects some workflows of the data processing. COOL DB structure is complicated: conditions data are spread over 30 Schemas and 10k Folders (about 1TB per data taking period). Every system change corresponds to new set of Folders. Long term COOL maintenance and evolution are problematic: COOL API (as well as CORAL) will be not supported by CERN IT Division after the beginning of Run3. COOL has no native support for the Global tags management.

1.2 Condition REST (CREST) data model

The data model in CREST consists of five tables which contain metadata and payload data. It is originally inspired by the CMS conditions DB [4]. Conditions data are stored in the PAYLOAD table. Values are consumed as an aggregated set (typically a header and some parameters container(s). Conditions meta-data are organized in three tables (plus one used essentially for mapping between tags and global tags) (see Figure 1).

IOV in CREST contains the time information, which is stored in one time column (time can be represented as a timestamp, a run number etc.) and it is valid by default until the next entry in time. An IOV points to one payload via an sha256 hash key. TAG in CREST is a label used to identify a specific set of IOVs. An additional table for tag metadata information was created to ease the migration of existing COOL data. GLOBAL TAG is a label used to identify a consistent set of TAGs, involved in a given data flow (e.g. a reprocessing campaign, a MC production etc). The same TAG can be associated to many GLOBAL TAGs.

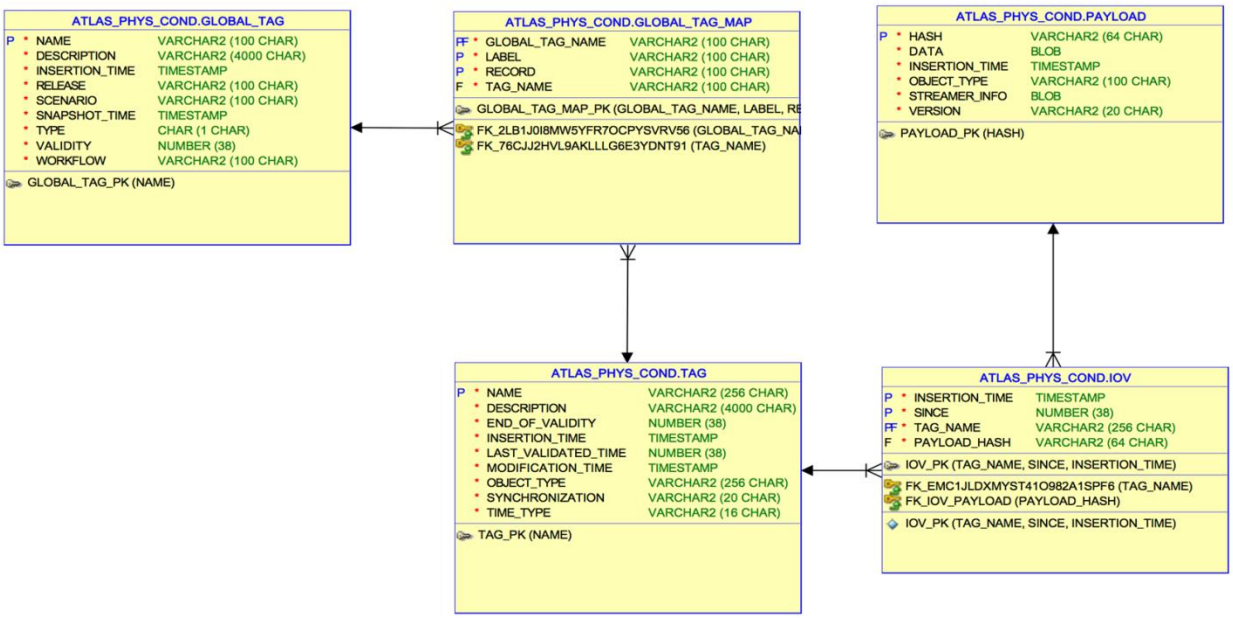

Figure 1. CREST data model scheme

The IOVs are retrieved separately from the PAYLOADS. This is a different behavior with respect to COOL, which loads everything (IOV+PAYLOAD) at the same time. This presents multiple advantages: first of all, the PAYLOAD access can be cached so it becomes faster, but also the IOVs can be checked very quickly, as they are de facto metadata of the PAYLOAD.

2. CREST Project Structure

The CREST project consists of several components: the CREST Server, the C++ Client library, the CREST Command Line Client and COOL to CREST Converter.

2.1 CREST Server

Like Frontier, the CREST DB server exposes functionalities via REST [5]. In the case of CREST, the API is described using OpenAPI specifications [6]. SQL is not involved in the dialogue between client and server, instead the internal resources are accessible via URLs. All HTTP verbs can be used: POST/PUT (to create/update resources), GET and DELETE. The request and response bodies are formatted in JSON. The header of the requests can be used for formatting the output, deal with caching related parameters etc. The prototype implementation is based on standard Java technologies (JEE, Spring) and specifications (JAX-RS [7], JPA), see Figure 2. The CREST server can be deployed in the same Tomcat server as Frontier or as a standalone service (using standard Java web servers like undertow, jetty, …).

The client library (in Python) and server stubs (in Java JAX-RS) are generated via OpenApi [8] by the Swagger [9] Codegen library. The first server version was implemented as a collaborative development within the ATLAS and CMS database teams.

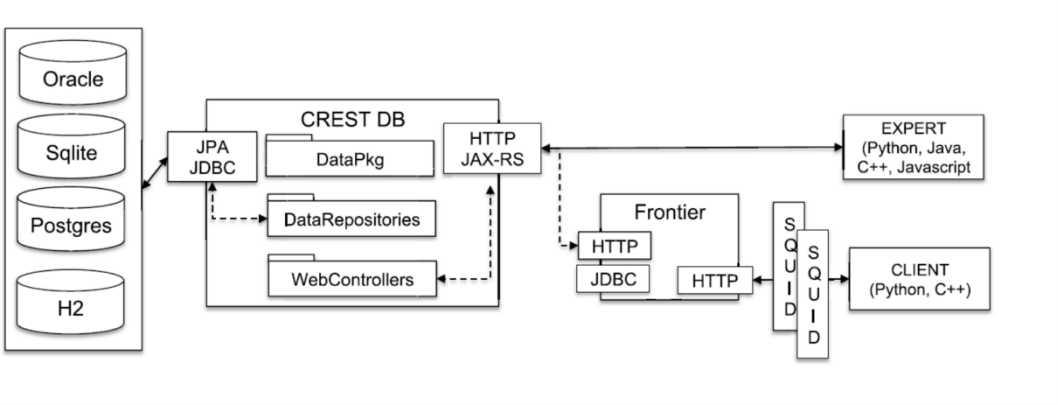

Figure 2. CREST Database Architecture

2.2 C++ CREST Client library (CrestApi library)

To simplify the CREST Server access to the C++ developers and the Athena framework, a C++ client library (CrestApi) was written. CrestApi library is a request library to the CREST Server (or to the local file storage). It allows to store, read (and update) the data on the CREST Server.

The data transferring mechanism (low level request methods) can be changed in the CrestApi library. The library prototype was created using the Boost Asio library. Now it uses the CURL [10] library.

The CrestApi library is written in C++, and the data exchanged with the server are in JSON format. The library uses the following external libraries: NLohmann JSON library [11], CURL, Boost Named Parameters library [12].

CREST Server API functions have many parameters, most of them are optional. CrestApi library uses the Boost Named Parameter Library to work with them. It allows to skip unused optional parameters when the method is called. Methods using the named parameters are in the CrestClientExt class (an extension of the standard CrestClient library).

2.3 CREST Command Line Client (CrestCmd)

The CREST command line client (CrestCmd) is an access tool to the data stored on the CREST server written to simplify the development of the other CREST project components. CrestCmd can be used for quick interactions with the CREST server, mainly with the goal to provide management functionalities and browsing capabilities to users. CrestCmd works with the main CREST data types such as tags, tag meta infos, global tags, global tag maps and an IOVs together with payloads. Each CrestCmd command has a built-in help.

2.4 COOL to CREST Converter

The CREST Converter is a command line tool to convert the existing data from COOL DB to CREST. After the CREST deployment, the converter will also be used as a tool for the existing user data conversion from COOL to CREST format. The CREST converter is based on the AtlCoolCopy/AtlCoolMerge tools and accepts the same parameters. If a connection with the CREST server fails, the CREST converter writes the data to disk in a format which allows later uploading to server. A cron job running the CREST converter will provide a background data transfer from COOL to CREST.

The essential differences between COOL and CREST data models require careful handling during the data conversion process. Each COOL IOV has two parameters defining the time interval: start and end time, CREST IOVs have the start time only. COOL folder has list of channels. The CREST data model does not know about channels. The information about the channels is stored inside the tag meta data.

COOL has no native API for global tag support. As a solution it is possible to add a special API for CREST implementation of the IDatabase interface of COOL. Now the algorithm to get the conditions data for a given global tag is organized as a chain: the logs from QTests are parsed to get a tag list, the tag list saved as a JSON file, then the data conversion for these tags starts.

3. Conclusion

The CREST project server prototype and its main components are implemented. The CREST C++ client library (CrestApi) was written and included in the official ATLAS software offline release (Athena). The conversion tool prototype to fill the CREST DB with the real data was realized and used to start filling CREST DB using COOL data; this allows for the creation and subsequent debugging of CREST-based applications for new users. The data conversion and data testing are running to optimize the CREST DB with the concrete conditions data.

References

[1] L.Rinaldi et al., Conditions evolution of an experiment in mid-life, without the crisis (in ATLAS) // EPJ Web Conf., Volume 214, 04052, 2019.

[2] ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, JINST 3 S08003 doi:10.1088/1748-0221/3/08/S08003, 2008.

[3] R.Trentadue et al., LCG Persistency Framework (CORAL,COOL, POOL): status and outlook in 2012 // J. Phys. Conf. Ser. 396 053067, 2012.

[4] Roland Sipos et al., Functional tests of a prototype for the CMS-ATLAS common non-event data handling framework // J. Phys. Conf. Ser. 898 042047, 2017.

[5] A.Formica A. and E.J.Gallas, A JEE RESTful service to access conditions data in ATLAS // J. Phys. Conf. Series 664 042016, 2015.

[6] OAI/OpenAPI Specification. Available at: https://github.com/OAI/OpenAPI-Specification (accessed 17.08.2021)

[7] JAVA JAX-RS. Available at: https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services (accessed 17.08.2021)

[8] OpenApi. Available at: https://www.openapis.org (accessed 17.08.2021)

[9] Swagger. Available at: http://swagger.io (accessed 17.08.2021)

[10] CURL Library. Available at: https://curl.se/docs/ (accessed 17.08.2021)

[11] JSON for Modern C++ - JSON for Modern C++. Available at: https://json.nlohmann.me (accessed 17.08.2021)

[12] The Boost Parameter Library. Available at: https://www.boost.org/doc/libs/1_67_0/libs/parameter/doc/html/index.html (accessed 17.08.2021)