Python object-oriented framework for consuming, manipulating and releasing non-event data for the CMS alignment and calibration

Joshua Dawes - CMS, CERN, The University of Manchester - joshua.dawes@student.manchester.ac.uk

Abstract

The Alignment, Calibrations and Databases group at the CMS Experiment delivers Alignment and Calibration Conditions Data to a large set of workflows which process recorded event data and produce simulated events. The current infrastructure for releasing and consuming Conditions Data was designed in the two years of the first LHC long shutdown to respond to use cases from the preceding data-taking period. During the second run of the LHC, new use cases were defined.

For the consumption of Conditions Metadata, no common interface existed for the detector experts to use in Python-based custom scripts, resulting in many different querying and transaction management patterns. A new framework has been built to address such use cases: a simple object-oriented tool that detector experts can use to read and write Conditions Metadata when using Oracle and SQLite databases, that provides a homogeneous method of querying across all services. The tool provides mechanisms for segmenting large sets of conditions while releasing them to the production database, allows for uniform error reporting to the client-side from the server-side and optimizes the data transfer to the server. The architecture of the new service has been developed exploiting many of the features made available by the metadata consumption framework to implement the required improvements.

This paper presents the details of the design and implementation of the new metadata consumption and data upload framework, as well as analyses of the new upload service's performance as the server-side state varies.

Overview

CondDBFW is used for…

- **SQL-free, object-oriented, instantiation-based querying of Conditions Metadata** in CMS web services, while using the web services' connection pools if necessary, and Python environments.
- **Verifying integrity of Conditions data** that has been released.
- **Adjustments of Conditions Metadata**, and the writing of scripts to automate fixing of future problem instances.
- **Simplifying Conditions Metadata querying** during Conditions release.

Example of Querying by Instantiation

IOVs can be copied from an Oracle database (through a Frontier layer) to a local SQLite database by…

Obtaining a connection object

```
from CondDBFW import shell
con = shell.connect() # Frontier
```

Instantiating a CondDBFW Conditions Metadata object ≡ selection of a row

```
tag = con.tag(name="test_tag")
```

Column to query — Type defines how to query

CondDBFW's internal type system allows using method chaining to process data

```
iovs = tag.iovs().data()
```

"Give me a wrapped list of *tag*'s IOVs. Then, unwrap that list to get the Python list of CondDBFW IOV objects."

Writing the new data to another database

```
sqlite_con = shell.connect("sqlite://db.sqlite")
sqlite_con.write_and_commit(iovs)
```

Conditions Release Architecture

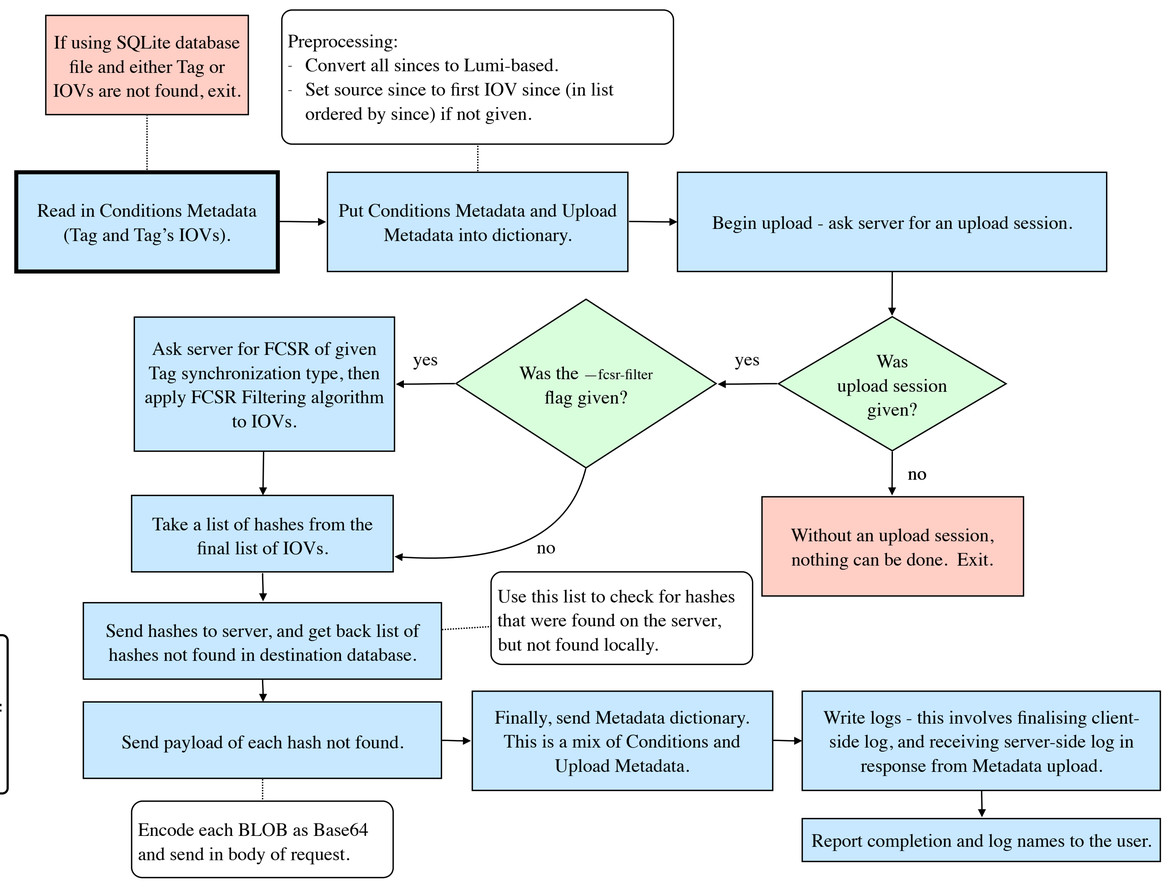
Split into a **client-side** and a **server-side** (both use **CondDBFW**);

The minimum amount of data is sent through each HTTPS request to the server, token-based authentication is used across requests;

The client-side is **deployed through a single script**, which copies the version of CondDBFW that the target upload server is using.

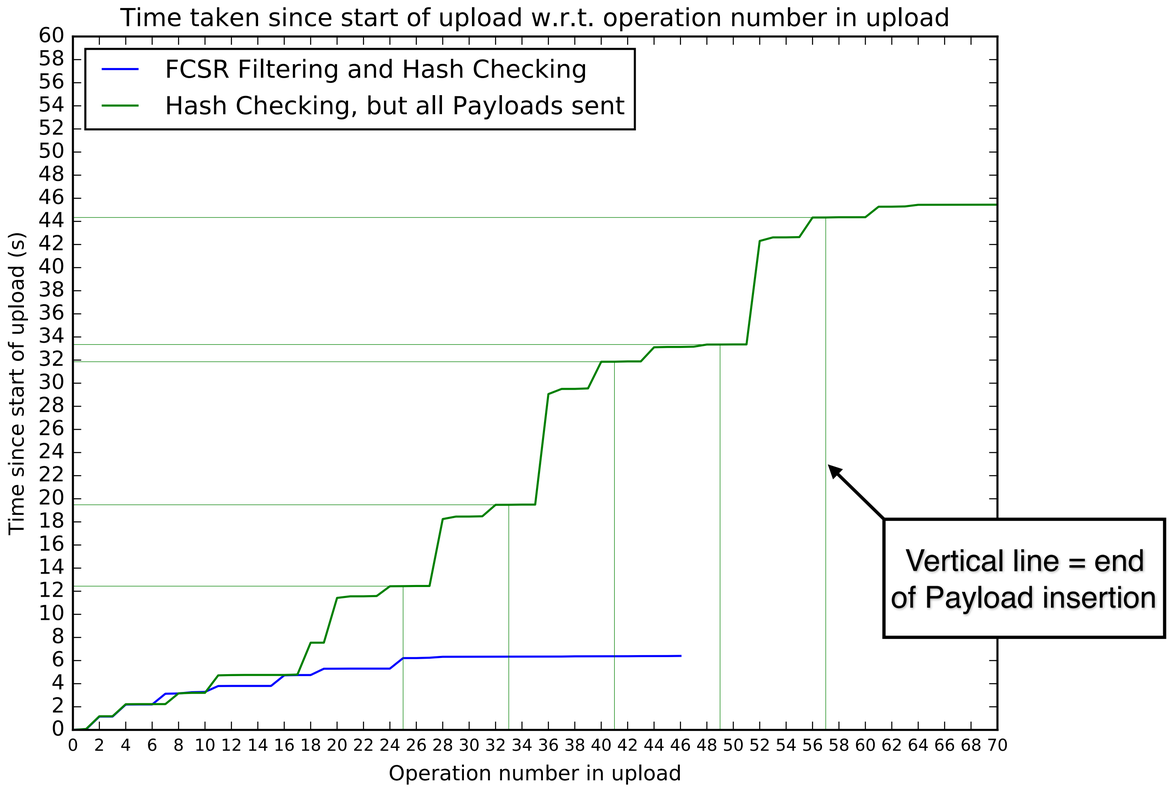
Upload Metadata

Values telling the client-side which Conditions to upload from the source given. This is required for the **Conditions Release Process:**



Optimisations and their Results

5 Payloads (≈ 255 MB of data) were sent to the Conditions database in the plot shown, using the **FCSR Filtering** and **Hash Checking** optimisations.



Hash Checking

For each Payload proposed for sending, only send if a query to the destination database for the Payload's hash yields no results.

Green line - 1st upload - all new Payloads, so needed to be sent ≈ 45 seconds.

Blue line - 2nd upload - no new Payloads ≈ 6 seconds.

Hence, Hash Checking reduced upload time by an order of magnitude.

FCSR (First Condition Safe Run) Filtering

FCSR = first run at which Conditions can be added without overwriting already stored (and frozen) Conditions.

Using the Conditions synchronization given by the —fcsr-filter flag, find the FCSR for that synchronization and filter IOVs on the client-side according to this FCSR.

Allows preparation of IOVs for insertion into a synchronization by validating them early.

Applied for the **blue line upload** and **filtered 5 Payloads down to 1**, hence FCSR Filtering is also an optimisation.

http://cms-alcadb-pub.web.cern.ch/cms-alcadb-pub/publications/2016/CHEP-Poster-cond-fw-upload.pdf

See CHEP 2016 Proceedings for the paper.