GRID 2021, Dubna

The development of a new conditions database prototype for ATLAS RUN3 within the CREST project

E.Alexandrov[1], A.Formica[2], M.Mineev[1], S.Roe[3]
(on behalf of the ATLAS Collaboration)

[1]Joint Institute for Nuclear Research, Dubna, Russia,
[2] Université Paris-Saclay, CEA/Saclay IRFU, 91191 Gif-sur-Yvette, IRFU/CEA, France,
[3] CERN, CH - 1211 Geneva 23, Switzerland

Outline

- Introduction:
  - Conditions Data
  - Motivations for a new conditions database
- CREST Data Model
- CREST Software
  - CREST DB Server
  - CREST C++ Client library (CrestApi)
  - CREST Command Line Client (CrestCmd)
  - COOL to CREST conditions data converter
- Conclusion

Conditions Data

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:

- detector calibration and alignment data,
- electrical and environmental measurements such as voltages, currents, pressures,
- temperatures,
- run and information about the data acquisition configuration,
- LHC beam information,
- trigger configuration,
- detector status data.

COOL Based Conditions 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.

COOL high level functionalities:

- 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.
  - Single-versioned Folders: IOVs can be appended (no overwrite, no *Tag* defined)
  - Multi-versioned Folders: IOVs can be stored in arbitrary way, Folder are Tagged (payload can be overwritten)
    - A Global Tag is a collection of multiple FolderTags
- Payloads are retrieved by providing IOV boundaries (and related Tag name)

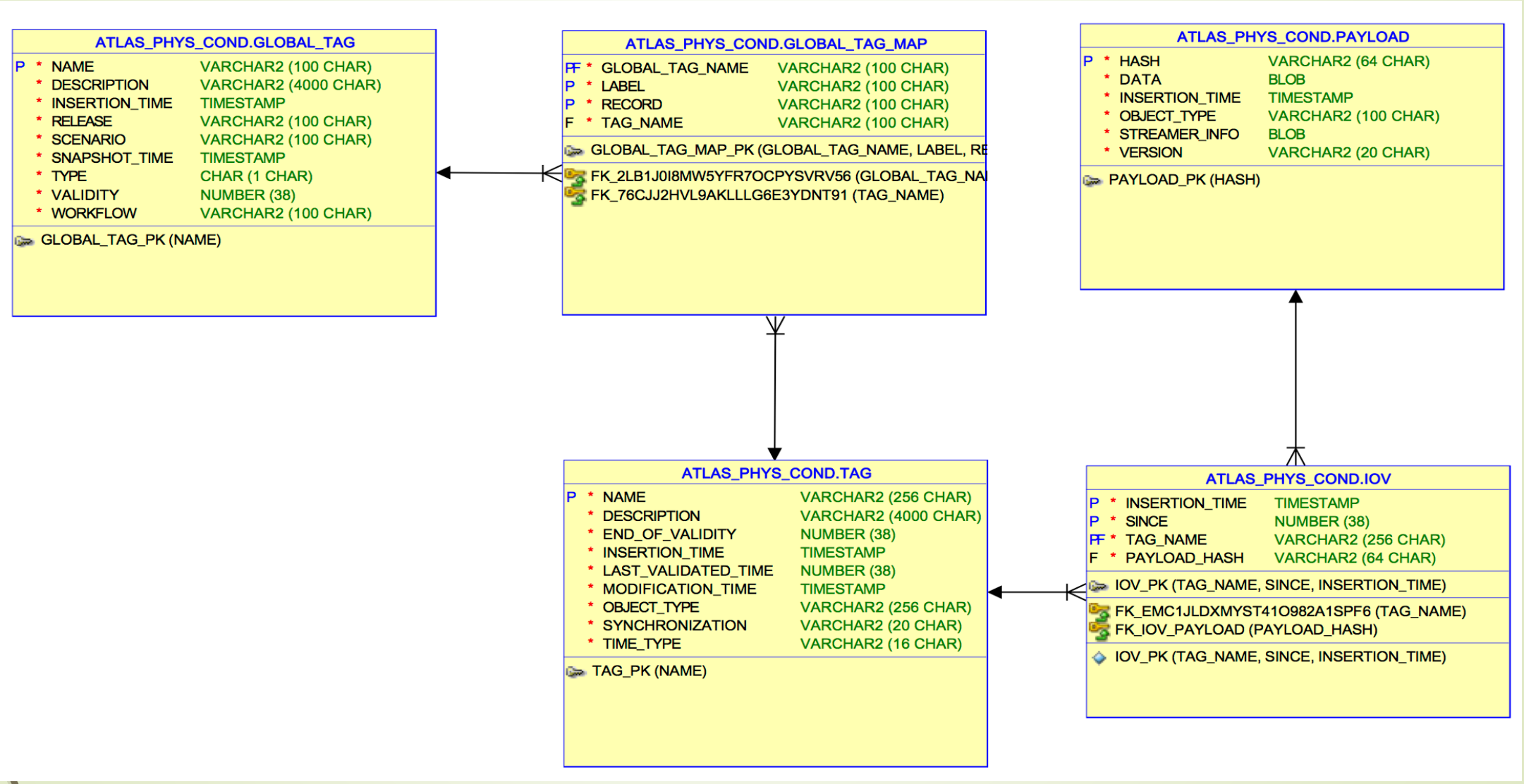
Motivation for a new database

- *Caching*: Conditions data payload loaded at the same time as the IOV (some workflows are badly cached!)
- *DB structure*: Conditions data spread over 30 Schemas and 10k Folders (about 1TB per data taking period). Every system change corresponds to new set of Folders
- Long term maintenance and evolution: COOL API (as well as CORAL) will be not supported by IT after the beginning of Run3.
- *Global tags management*: there is no native support in COOL)

Condition REST (CREST) data model

- The data model consists of five tables which contain metadata and payload data. It is originally inspired from the CMS conditions DB.
- **Conditions data**: they are stored in the PAYLOAD table. Values are consumed as an aggregated set (typically a header and some parameters container(s).
- **Conditions meta-data**: they are organised in three tables (plus one used essentially for mapping between tags and global tags)
  - IOV: 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: 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: a label used to identify a consistent set of TAGs, involved in a given data flow. A TAG can be associated to many GLOBAL TAGs.

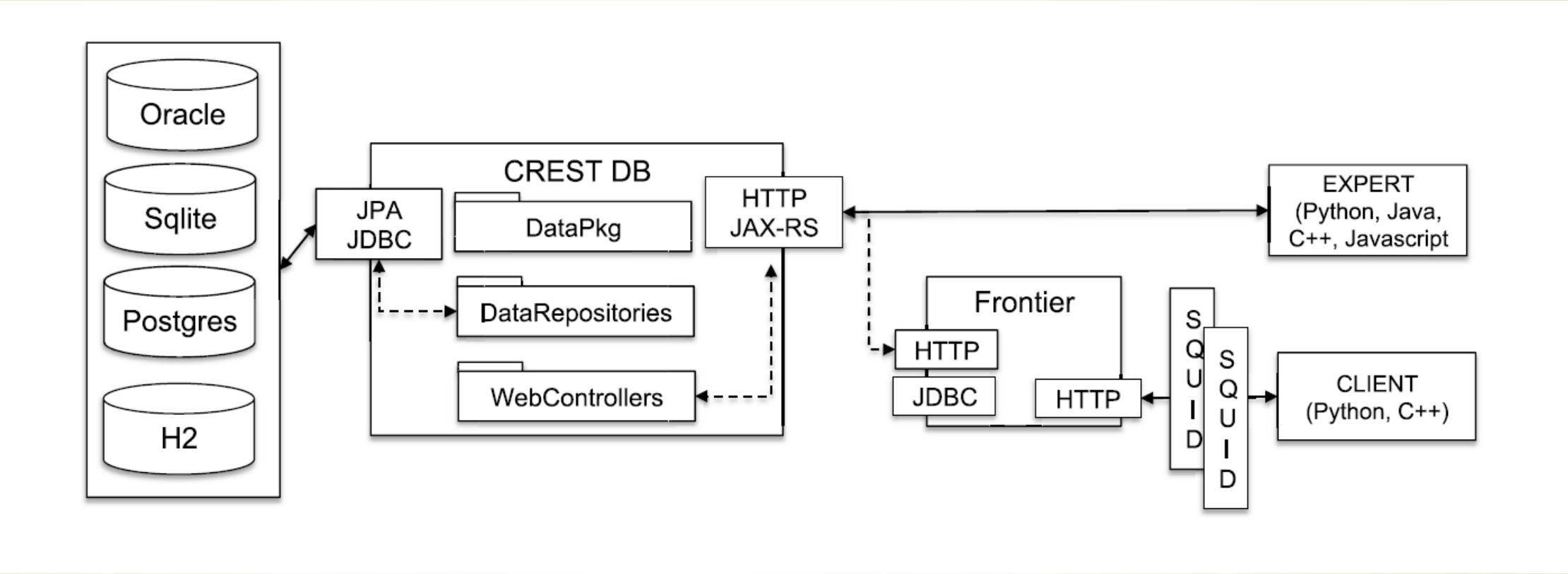
CREST data model scheme



Crest DB server

- **As Frontier, the CrestDB server exposes functionalities via REST**
  - But *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…
    - GET <server>/tags/A_TAG_NAME => retrieve resource with id A_TAG_NAME
    - POST <server>/tags {request body} => create new resource (parameters are taken from body)
    - PUT <server>/tags/A_TAG_NAME {request body} => update resource A_TAG_NAME
    - DELETE <server>/tags/A_TAG_NAME => delete resource A_TAG_NAME
  - Request and Response bodies are formatted in JSON
  - Header of the requests can be used (e.g. for formatting the output, deal with caching related parameter etc.)
- **Prototype Implementation**
  - Based on standard Java technologies (JEE, Spring) and specifications (JAX-RS, JPA)
  - Can be deployed in the same Tomcat server as Frontier or as a standalone service (using standard java web servers like undertow, jetty, …).

CREST Database Architecture



The CREST API has been written using OpenAPI specifications. The client library (in Python) and server stubs (in Java JAXRX) are generated via OpenApi Codegen library. The first version was implemented as a collaborative development within ATLAS and CMS database teams.

CrestApi Library

- 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 can be changed in the CrestApi library. (The library prototype was created using Boost Asio library. Now it uses CURL library.)

- CrestApi library is written in C++, and the data exchanged with the server are in JSON format. The main dependencies are:
  - CURL library (to communicate with server),
  - Boost Library (boost named parameters),

Optional parameters in the CrestApi methods

- CREST Server API functions have many parameters, most of them can be optional. CrestApi library uses the Boost Named Parameter Library to work with them.

- Example (IOV list method):

```
nlohmann::json list1 = myCrestClient.findAlliovsParams("myTag");
nlohmann::json list2 = myCrestClient.findAlliovsParams("myTag",5,3); // here: _page=3,_size=5
nlohmann::json list2 = myCrestClient.findAlliovsParams("myTag",_page=3,_size=5);
nlohmann::json list2 = myCrestClient.findAlliovsParams("myTag",_sort="id.since:ASC",_page=3,_size=5);
```

It is possible to skip some 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).

CREST Command Line Client (CrestCmd)

The command line client is intended to browse the data stored on the CREST server to simplify the development of the other CREST project components (check if the data exist or to insert them for tests).

- CREST Command Line Client (CrestCmd) can be used for quick interactions with CREST server, mainly with the goal of provide management functionalities and browsing capabilities to users.

CrestCmd Examples:

- Get command list:

  crestCmd get commands

- Get a tag with the name ***test_MvG3***:

  crestCmd get tag -n test_MvG3

- Get a command description for ***get tag*** command:

  crestCmd get tag –h

CrestCmd commands:

- **Get list methods** consists of the methods to get the lists of **tag, global tags, global tag maps** and **IOVs**
- **Get methods** consists of methods to find a **concrete tag, tag meta info, global tag, payload** and **payload meta info**.
- **Create methods** consists of methods to create a **tag, tag meta info, global tag, global tag map** and **an IOV together with a payload**.
- **Remove methods** consists of methods to **remove global tag** and **tag**.

CREST Converter: Requirements

- Command line Tool
  - During prototyping, the converter is used to produce CREST datasets from existing COOL data
  - In deployment, it provides a tool to allow users to copy their existing data to CREST
  - It should be able to accept the same parameters as the current AtlCoolCopy/AtlCoolMerge tools and be user friendly
- Backup data
  - If the CREST connection momentarily fails, write the data to disk in a format which allows later upload to CREST
- Cron job for converting COOL to CREST (not yet realized)
  - Run in the background to continually upload data from COOL to CREST

CREST Converter

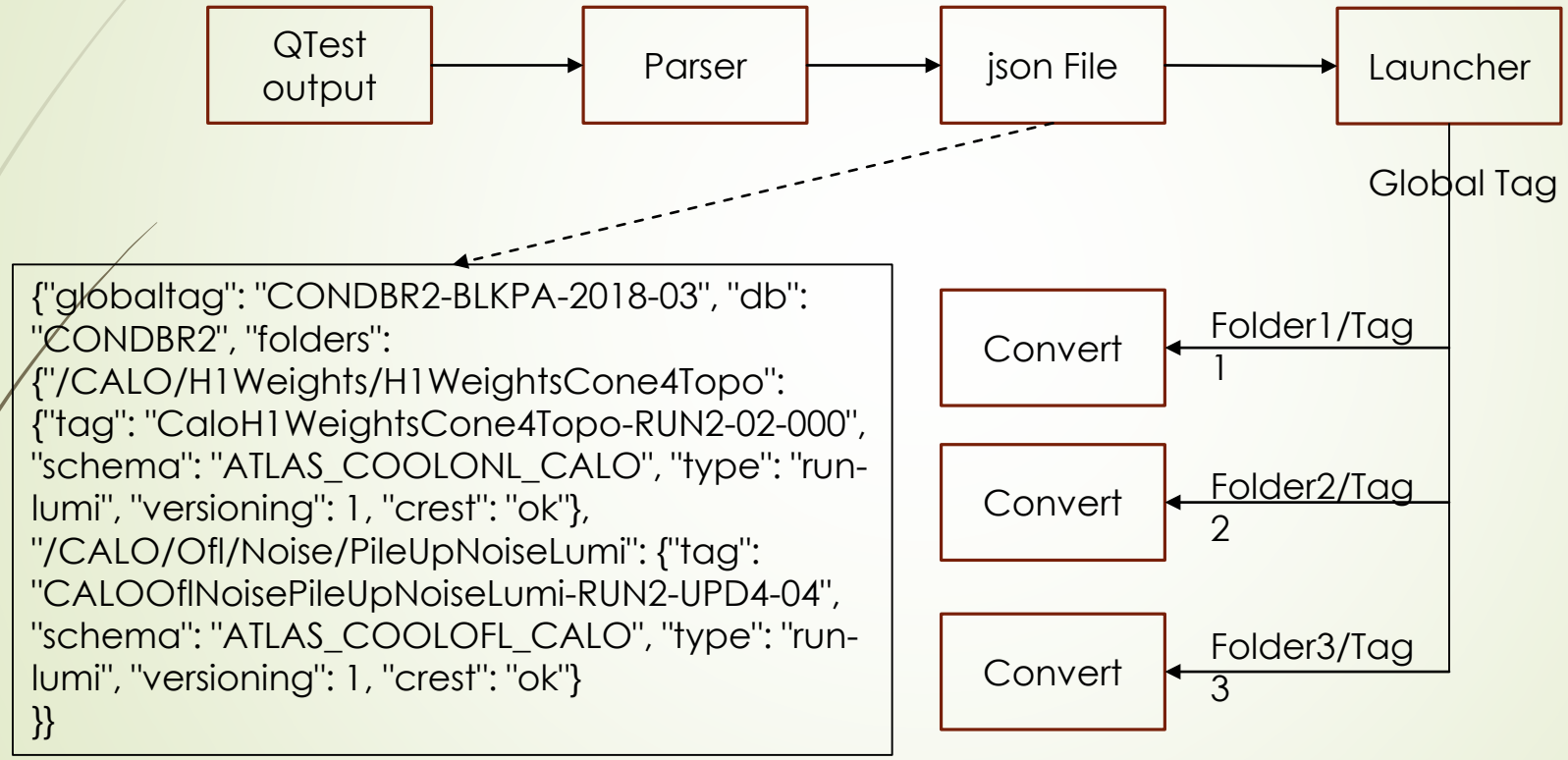
Main differences between COOL and CREST data models which require careful handling during the migration process.

- **COOL data format**:
  - IOV has start and end time
  - Folder has set of channels and IOVs
  - Each channel can have its own set of IOVs
- **CREST data format**:
  - IOVs only have a start time
  - Folder has no channels. It has a set of IOVs only. Each IOV has data in JSON format with all channels. CREST data model does not know about channels.

CREST Converter: global tag

- Problem:
  - COOL has no native API for global tag
- Solution:
  - Add special API for CREST implementation of IDatabase interface of COOL,
  - get a global tag list from QTest output with the corresponding tags,
  - run data conversion for all these tags,

GlobalTag data conversion Scheme



Conclusion

- The CREST project prototype is implemented.
- The CREST C++ client library (CrestApi) was written and included in the official 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, thus allowing to tests further software components in Athena.