10th International Conference "Distributed Computing and Grid Technologies in Science and Education" (GRID'2023)

Evolution of the ATLAS CREST Conditions Database Project

E.Alexandrov, A.Formica, M.Mineev, N.Ozturk,

S.Roe, V.Tsulaia, M.Vogel

Outline

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.

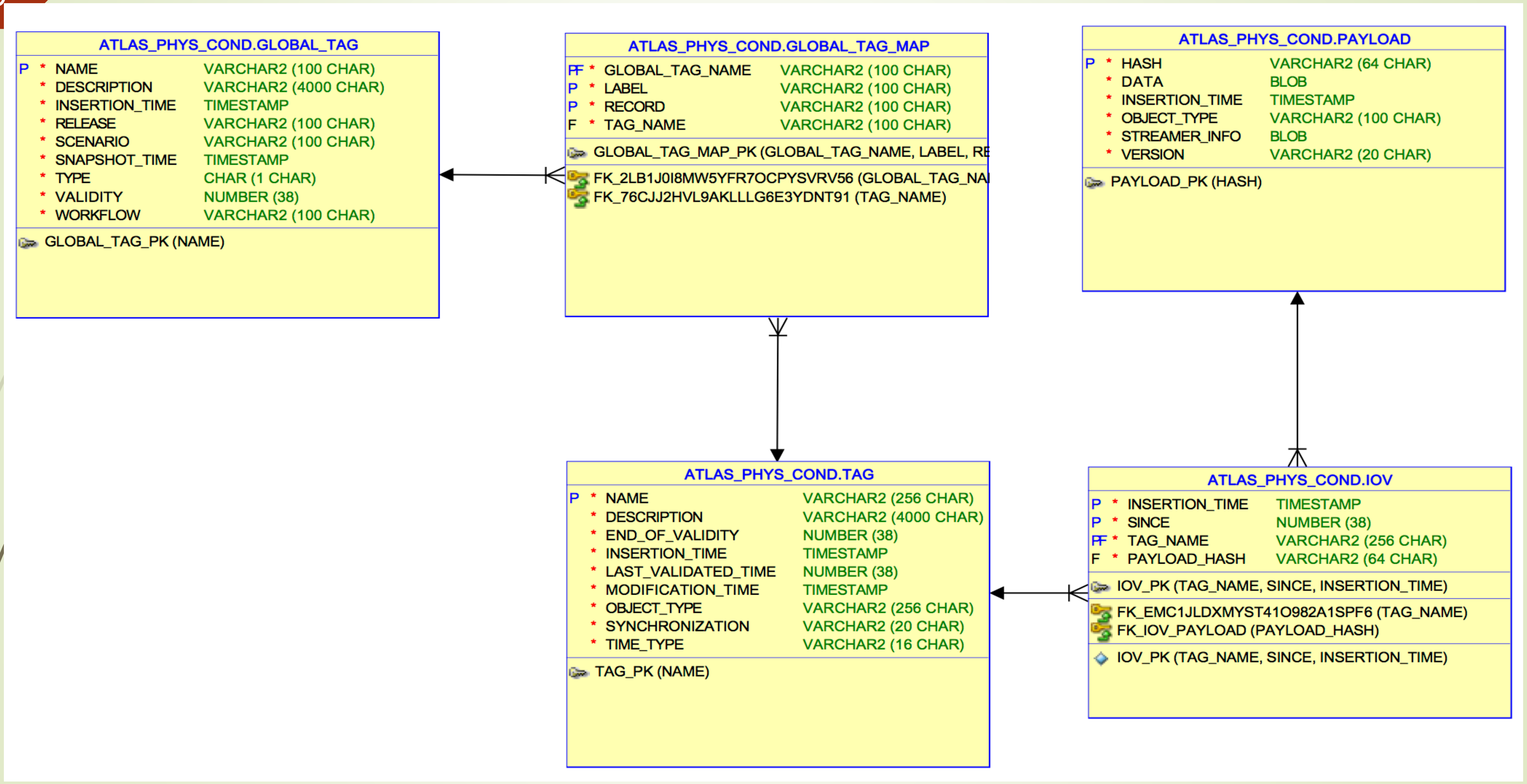
Motivation for a new database

- *Caching*: Conditions data payload loaded at the same time as the IOVs in the current model COOL (data for some workflows are badly cached!)
- Long term maintenance and evolution for the COOL API and CORAL software were a concern.
- *Global tag management*: there was no native support in COOL.

Conditions 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 database.
- **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



ATLAS_PHYS_COND.GLOBAL_TAG

P	*	NAME	VARCHAR2 (100 CHAR)
	*	DESCRIPTION	VARCHAR2 (4000 CHAR)
	*	INSERTION_TIME	TIMESTAMP
	*	RELEASE	VARCHAR2 (100 CHAR)
	*	SCENARIO	VARCHAR2 (100 CHAR)
	*	SNAPSHOT_TIME	TIMESTAMP
	*	TYPE	CHAR (1 CHAR)
	*	VALIDITY	NUMBER (38)
	*	WORKFLOW	VARCHAR2 (100 CHAR)

GLOBAL_TAG_PK (NAME)

ATLAS_PHYS_COND.GLOBAL_TAG_MAP

PF	*	GLOBAL_TAG_NAME	VARCHAR2 (100 CHAR)
P	*	LABEL	VARCHAR2 (100 CHAR)
P	*	RECORD	VARCHAR2 (100 CHAR)
F	*	TAG_NAME	VARCHAR2 (100 CHAR)

GLOBAL_TAG_MAP_PK (GLOBAL_TAG_NAME, LABEL, RE

FK_2LB1J0I8MW5YFR7OCPYSVRV56 (GLOBAL_TAG_NA

FK_76CJJ2HVL9AKLLLG6E3YDNT91 (TAG_NAME)

ATLAS_PHYS_COND.PAYLOAD

P	*	HASH	VARCHAR2 (64 CHAR)
	*	DATA	BLOB
	*	INSERTION_TIME	TIMESTAMP
	*	OBJECT_TYPE	VARCHAR2 (100 CHAR)
	*	STREAMER_INFO	BLOB
	*	VERSION	VARCHAR2 (20 CHAR)

PAYLOAD_PK (HASH)

ATLAS_PHYS_COND.TAG

P	*	NAME	VARCHAR2 (256 CHAR)
	*	DESCRIPTION	VARCHAR2 (4000 CHAR)
	*	END_OF_VALIDITY	NUMBER (38)
	*	INSERTION_TIME	TIMESTAMP
	*	LAST_VALIDATED_TIME	NUMBER (38)
	*	MODIFICATION_TIME	TIMESTAMP
	*	OBJECT_TYPE	VARCHAR2 (256 CHAR)
	*	SYNCHRONIZATION	VARCHAR2 (20 CHAR)
	*	TIME_TYPE	VARCHAR2 (16 CHAR)

TAG_PK (NAME)

ATLAS_PHYS_COND.IOV

P	*	INSERTION_TIME	TIMESTAMP
P	*	SINCE	NUMBER (38)
PF	*	TAG_NAME	VARCHAR2 (256 CHAR)
F	*	PAYLOAD_HASH	VARCHAR2 (64 CHAR)

IOV_PK (TAG_NAME, SINCE, INSERTION_TIME)

FK_EMC1JLDXMYST41O982A1SPF6 (TAG_NAME)

FK_IOV_PAYLOAD (PAYLOAD_HASH)

IOV_PK (TAG_NAME, SINCE, INSERTION_TIME)

CREST server

- **The CREST server realizes REST API**
  - 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 format
  - Header of the requests can be used (e.g. for formatting the output, deal with caching related parameter, OAuth2 authentication etc.)
- **CREST server 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 server is compatible with multiple DB technology (e.g.: Oracle, PostgreSQL).

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),
  - NLohmann JSON Library,
  - Boost Library (boost named parameters).

https://gitlab.cern.ch/atlas/athena/tree/master/Database/CrestApi

https://gitlab.cern.ch/crest-db/CrestApi

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).

OAuth2 authentication in CrestApi

CREST project will have the OAuth2 authentication support. Now we work on the prototype.

CREST server with OAuth2 authentication:

https://crest-03.cern.ch:8443/api

New **CrestApi** prototype with the OAuth2 support:

https://gitlab.cern.ch/crest-db/CrestApi/-/tree/oauth2

Setup for OAuth2:

To switch on the OAuth2 authentication it is necessary to set a variable:

s_CREST_AUTH = true in the **./CrestApi/CrestApi.h**:

```
inline static const bool s_CREST_AUTH = true;
```

CrestApi uses the **OAuth2 token**. This token can be received by the script:

```
../CrestApi/scripts/curl-gettoken.sh
```

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

crestCmd and Utilities in git:
https://gitlab.cern.ch/crest-db/crest-cmd

crestCmd utilities

crestImport – import data from local file storage to the CREST server.

crestExport – export data from CREST server to the local file storage.

(data: tag, tag meta info, IOVs, payloads)

crestCopy – create a tag copy with a new name on the CREST server.

tag_creation – create dummy data on the local file storage.

removeTagList - remove a global tag with all its tags.

COOL to CREST converter

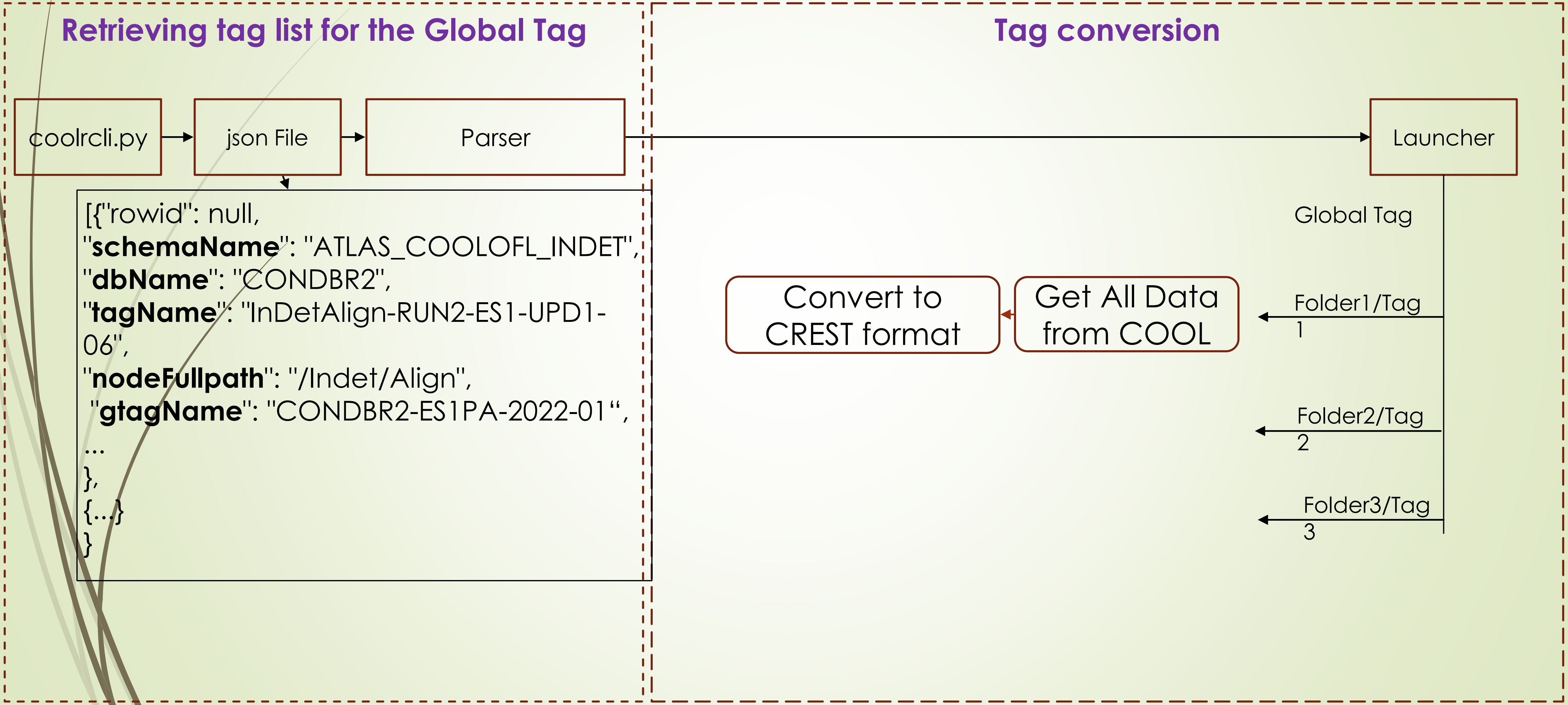
Basic properties

- Converter based on the AtlCoolCopy and supports most of its additional options (example of options: set folders or set tags or set IOVs)
- Support for all types of COOL data
- Support for converting all data from Global Tag
- Can do backup (if connection to server is broken or do not exist)
- Has a tag summary for analyzing the time and result of uploading the selected tag.
- Has a Global Tag summary to check the uploading status of all tags from the Global Tag

It uses C++ client library (CrestApi) for CREST.

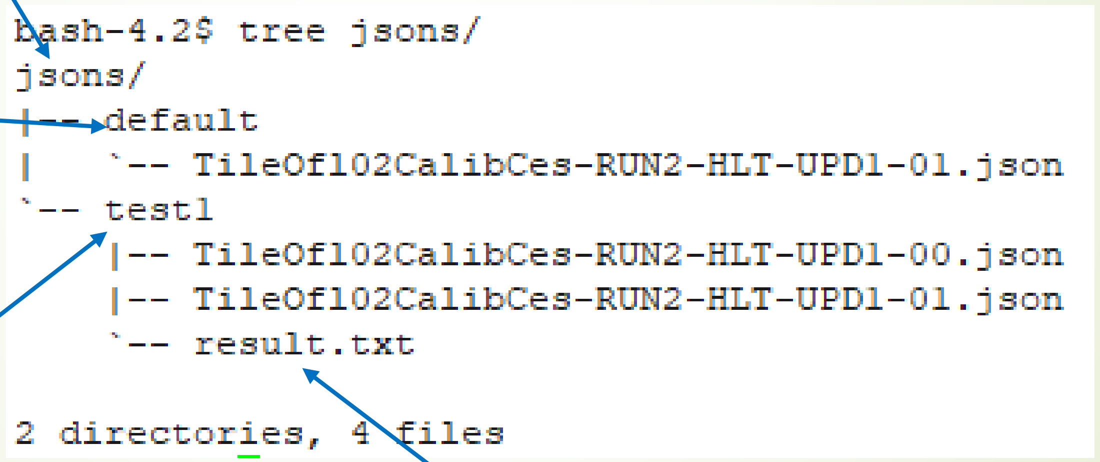
git: https://gitlab.cern.ch/crest-db/crestconverter

Global Tag data conversion



```
[{"rowid": null,
"schemaName": "ATLAS_COOLOFL_INDET",
"dbName": "CONDBR2",
"tagName": "InDetAlign-RUN2-ES1-UPD1-
06",
"nodeFullpath": "/Indet/Align",
 "gtagName": "CONDBR2-ES1PA-2022-01“,
…
},
{…}
}
```

Summary output structure



```
bash-4.2$ tree jsons/
jsons/
|-- default
|   `-- TileOfl02CalibCes-RUN2-HLT-UPD1-01.json
`-- test1
    |-- TileOfl02CalibCes-RUN2-HLT-UPD1-00.json
    |-- TileOfl02CalibCes-RUN2-HLT-UPD1-01.json
    `-- result.txt

2 directories, 4 files
```

New subfolder for summary

Subfolder for summary without Global Tag

Subfolder for summary with "test1" Global Tag

Summary for tag

Summary for Global Tag "Test1"

Library to prepare payload data in JSON format (crest_jsondata)

Requirements:
We should define a new package which can be used by system experts in their code to save data for each channels. Then this package should be able to dump the content of this generic structure into a JSON CLOB as it does now for COOL2CREST migration. The separate package can later on be used in the converter as well for testing purpose.

git: https://gitlab.cern.ch/crest-db/crest_jsondata

First client is PVSS2CREST (put archived DCS data in CREST)

crest_jsondata library: API

Constructor:

```
    CondContainer();
```

Structure of data (fields), they should be set before data:

```
    void addColumn(const char* name, typeId type);   // "name" is name of field
    void addColumn(const char* name, uint32_t type); // "type" is type of field
```

Set time:

```
    void setSince(uint64_t since);
```

Add data:

//put all data with channel

```
    void insertChannel( char* chanName,...); // "chanName" is name of channel
```

//store data

```
    void storeData(uint32_t count,...);     // "count" – the number of parameter
    void insertChannelFromMemory(const char* chanName);
```

Get results as JSON:

```
    nlohmann::json getJson();
```

CREST data and Athena

Testing an Athena job by reading data from CREST: https://its.cern.ch/jira/browse/ATDBOPS-185

A job from Liquid Argon (LAr) calorimeter. The jobOption:

../athena/LArCalorimeter/LArROD/python/LArRawChannelBuilderCrestConfig.py

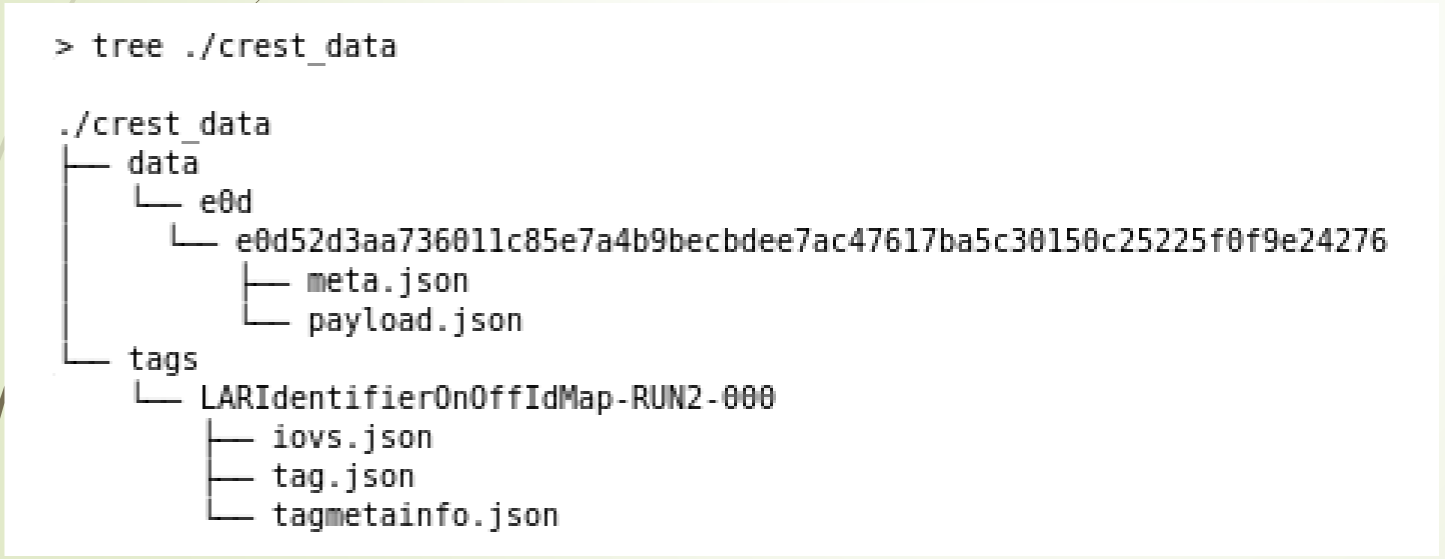
CREST options in Python:

```
...
 acc.getService("IOVDbSvc").GlobalTag="CREST-RUN12-SDR-25-MC"
 acc.getService("IOVDbSvc").Source="CREST"          # use CREST data
 acc.getService("IOVDbSvc").OutputLevel=DEBUG
 acc.getService("IOVDbSvc").CrestToFile = True        # CREST data dump
...
```

CREST data dump from jobOption

The option in Python to dump the data:

```
...
acc.getService("IOVDbSvc").CrestToFile = True           # CREST data dump
...
```



```
> tree ./crest_data

./crest_data
├── data
│   └── e0d
│       └── e0d52d3aa736011c85e7a4b9becbdee7ac47617ba5c30150c25225f0f9e24276
│           ├── meta.json
│           └── payload.json
└── tags
    └── LARIdentifierOnOffIdMap-RUN2-000
        ├── iovs.json
        ├── tag.json
        └── tagmetainfo.json
```

Next steps

- Integration of CREST in Athena framework (changes in IOVDbSvc, IOVDbFolder,…).
- Data migration for different ATLAS subsystems (LAr, Tile, HLT,…)
- DCS data handling in CREST.
- OAuth2 authentication in the production versions.
- …

Thank you!