Conditions Database for the Mu3e Experiment

Urs Langenegger

Mu3e Internal Note 0xxx

1st September 2023

This note describes a complete implementation of a conditions database (CDB) for the Mu3e experiment. The CDB architecture is completely independent of the underlying database engine and three different server implementations are provided. The design principles, the changing conditions management, the data model and software, and the storage setup are discussed. The tools required for CDB interaction and synchronization between different server implementations are described. As a use cases, we describe the CDB for the 2023 data challenge, on the one hand using data of the 2020 detector integration run (with pixel data only), and on the other hand using data from Monte Carlo simulation (with data from all detector components and changed pixel configurations).

Contents

Document history:

Ver.	Date	Author	Comment
1.0	September 4, 2023	Urs	Initial version

1 Introduction

The data measured by the Mu3e experiment comprises event-data (the data resulting from interactions of charged particles with the sensitive elements of the Mu3e detector), environmental data (for instance, temperature), and operational data (*e.g.*, beam current and magnetic field strength). The per-event data recorded by the Mu3e experiment are subject to time-dependent artefacts such as

- defective (noisy or dead) channels, detector or readout elements components;
- changing positions of detector components;
- preliminary calibrations, potentially to be superseeded with improved calibrations derived with larger datasets;

These artefacts can be corrected for, or at least taken into account, with calibrations[1].

Managing the calibrations has two aspects: On the one hand, (reconstruction or analysis) code needs access to the correct calibration constants for the event-data at hand, matching the software version, and on the other hand, these calibration constants need to be fetched from somewhere. These calibrations (algorithms and constants) represent non-event data that are stored separately from the event data, in a conditions database (CBD).

Here we present a "schemaless" implementation of a CDB that is completely independent of the chosen database (DB) technology. Traditionally, a database schema describes the organization and (logical) relationship of various data entities in a database. Despite the purported benefits of database schemata (mostly related to controlling access and data integrity), the majority of modern high-energy physics experiments have moved away from this database paradigm and rather implemented schemaless databases where all data is stored in documents containing a (minimal) schema within themselves and with no (or minimal) connections between documents.

For discussions about modern conditions databases in high-energy physics for, *e.g.*, Belle [1], CMS [2], ATLAS+CMS [3], LHCb [4], or from the HEP Software Foundation [5]. Within the Mu3e collaboration, databases (also beyond the conditions database) have been discussed sporadically, *e.g.*, in Ref. [6]. The SpecBook also contains a section on this topic [7].

The setup presented here constitutes a complete and working framework. It is not optimized (yet) for minimal DB storage or for memory usage inside running processes.

2 Design principles

In this section we first describe the design of the database structure storing calibration data before describing the software providing access to the calibration data.

[1] In the context of this document, alignment is considered a calibration.

2.1 Conditions database

The event-data recorded in runs with specific detector configurations (fixed during a run) require calibration constants with an interval of validity (IOV) that is a priori not coupled to single runs and, indeed, should cover longer periods of multiple runs (stable conditions should be the ideal). Figure 1 illustrates this situation. By definition, a "tag" is the combination of a specific calibration (comprising both algorithms and constants), identified through its name and possibly version, with a list of IOVs. By choice, an IOV is open-ended, *i.e.*, the correct IOV for a given run X is the IOV labeled with the largest run number smaller than X. For example, given a tag with IOV list $[1, 236, 567]$, the correct IOV for run 235 would be IOV 1.

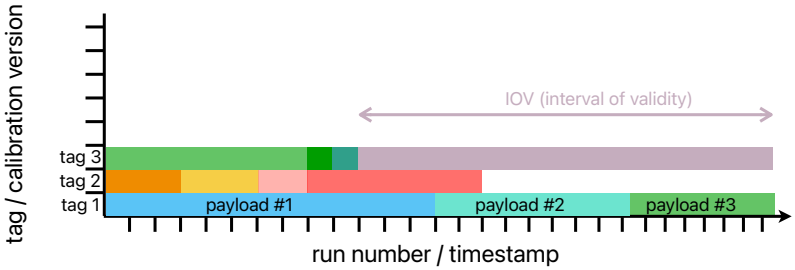

Figure 1: Changing conditions for calibration payloads that need to be managed through the CDB. The time-dependence on the abscissa is indicated through the run numbers, while the tags on the ordinate indicate different calibrations (either different, improved, versions of a calibration, a different type of calibration for the same detector components, or calibrations for different detector components). A payload consists of the calibration constants and descriptive meta-data. Each payload has a specific interval of validity, which is (by design choice) open-ended.

Processing the event-data of a specific run requires the constants for (possibly many) calibrations of all involved detector components, possibly for multiple software versions (in case the data of different data taking periods require different software versions). To manage this complex situation, "global tags" provide the means to organize tags for specific software versions and tasks: A global tag contains a list of tags.

For the purpose of storage, the calibration constants are transformed into payloads consisting of descriptive meta-data and and actual data in the form of (eventually[2]) a binary large object (BLOB). A "hash"[3] uniquely labels a payload (given a tag and IOV) in the storage backend, for example for a "pixel quality" calibration payload of the `mcideal` global tag

```
hash = tag_pixelquality_mcideal_iov_1
```

[2]At the moment, the calibration data is not stored in binary form, but rather base64 encoded. This is an implementation detail that does can be changed without affecting the CDB design.

[3]At the moment, the hash is a simple string.

Here, `tag` and `iov` are unchanging components of the has, while `pixelquality`, `mcideal`, and `1` indicate variable components.

The design of the database *storage* is kept very simple and is illustrated in Fig. 2. There are three required collections (or tables, in SQL[4] terminology) storing everything related to the conditions data. The three required collections are the "global tags", the "tags" (or "iovs"), and the "payloads". In addition, there might be other collections for auxiliary data, *e.g.*, the run database (collecting basic information on run like start and end times, number of events, readout configuration, *etc.*) or meta-data (for easier database access).

It should be noted that there is no unique "key" (`e.g.`, the runnumber) to "everything". Global tags and tags have their names as key, while the payloads are keyed with the hash.

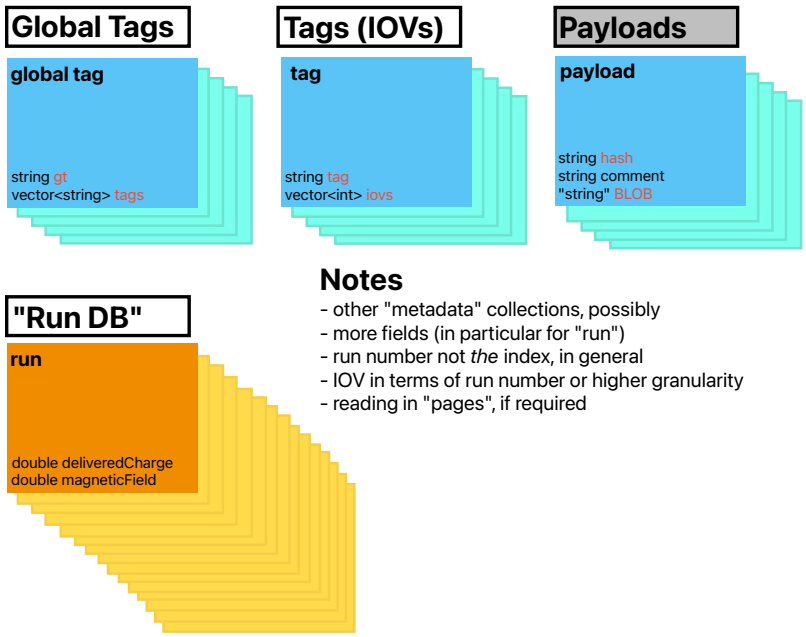

Figure 2: Data model of the conditions database.

2.2 Access to the conditions data

To apply a calibration to (reconstruction) event-based data requires (1) code ("calibration algorithm") and (2) data ("calibration constants"). The code will be particular to specific detector components. The data access is unified in the abstract base class `calAbs`, while the interpretation of the data is implemented in derived classes.

The conditions data need to be accessed (written or read) in different settings. Inside the Mu3e (reconstruction) code, the central access point to conditions data is a singleton class `Mu3eConditions` that provides

[4]Structured Query Language, often used in the context of schema-based databases.

- access to the database through an abstract base class `cdbAbs`. Three concrete server version are currently provided: (1) filesystem based JSON files, (2) local MongoDB server, and (3) REST API cloud based ATLAS MongoDB server.
- the organization and cached access to all global tags and tags/IOVs in the CBD.
- a map of all registered calibrations. Each calibration class is stored as pointer to `calAbs` that keeps track of its initialization database.

The derived calibration classes (*e.g.*, calPixelAlignment, calPixelQuality or calPixelCablingMap) provide the interpretation (both in terms of reading and writing) of the BLOB data into the concrete calibration constants. The derived classes are the *one and only* place interfacing code and database contents. In a sense it is here that the document schema is implemented.

Figure 3: Class organization for Mu3e conditions management and access to database storage and calibration constants.

3 Data model and software architecture

In this section the code illustrated in Fig. 3 is described in more detail. Both Mu3e reconstruction code and outside CDB-specific code is covered.

3.1 Conditions management

The central access to anything related to conditions is encoded in a singleton class `Mu3eConditions` that connects calibrations code to the conditions database(s). It is a singleton class because there should be one well-defined place only to obtain calibration constants in the Mu3e (reconstruction) code. The `Mu3eConditions` object is instantiated with a specific global tag and database pointer. For this default setup, `Mu3eConditions` organizes the tags (IOVs) for

this specific global tag and provides the correct calibration constants for any given run number.

`Mu3eConditions` requires all calibrations to be registered for proper serving of the correct calibration constants (correct both in terms of calibration algorithm and IOV). Whenever conditions change (for instance, a reconstruction job receives events with a new runnumber), `Mu3eConditions` notifies each registered calibration with associated tag to check whether its IOV is still valid and, if not, load a new payload from the database.

The calibration constants are determined from such database payloads and stored and accessed in derived classes of the abstract base class `calAbs`.

3.2 Calibration classes

The abstract base class `calAbs` provides

For each instantiation, this class keeps a cache map of a hash key and payload value. The hash provides a unique identification (string) of the payload, based on the tag and the IOV (see section 2.2). The common functionality provided by `calAbs` is limited to payload reading and writing, without *interpretation* of the contained BLOB, and the determination whether reading a new payload is required, given a hash (*i.e.* an IOV, which depends on the run number).

The payload interpretation is only possible with detailed knowledge of the contained data and is performed in the derived classes, *e.g.*, calPixelAlignment, calPixelCablingMap, or calPixelQuality[5].

Obtaining the payload is deferred to the concrete DB implementation classes. In addition, `calAbs` implements payload read/write operations for file I/O. In each concrete DB implementation class, the calibration constants are stored in a

```
std::map<uint32_t, constants> fMapConstants;
```

where the `struct constants` is different for each implementation class.

The class `calPixelAlignment` (and the classes `calFibreAlignment`, `calMppcAlignment`, as well as `calTileAlignment`) provides the CDB equivalent to the information in the corresponding trees in the `alignment` TDirectory of a Mu3e simulation output rootfile. The constants stored are

```
struct constants {
  uint32_t id;
  double vx, vy, vz;
  double rowx, rowy, rowz;
  double colx, coly, colz;
  int nrow, ncol;
  double width, length, thickness, pixelSize;
};
```

To access the information stored in the calibration constants, direct accessors are provided

[5]The discussion is focused on pixel calibrations. Alignment classes have been set up for all detector components (fibres, tiles, MPPCs) as contained in the `alignment` TDirectory of a Mu3e rootfile of a simulation run

```
uint32_t id(uint32_t id) {return fMapConstants[id].id;}
double vx(uint32_t id) {return fMapConstants[id].vx;}
double vy(uint32_t id) {return fMapConstants[id].vy;}
// ...
double pixelSize(uint32_t id) {return fMapConstants[id].pixelSize;}
```

These accessors can be used (for instance) in `mu3eTrirec/src/SiDet.cpp`. Since the constants, and therefore their accessors, are implementation class dependent, the `calAbs` pointer obtained from `Mu3eConditions` has to be cast to the derived class.

```
void SiDet::readSensors(Sensor::map_t& sensors, TTree* tree) {
    if(!tree) {
        mu3e::log::warn("[SiDet::readSensors] No sensors alignment tree
            found (tree == NULL).\n");
        return;
    }
    Mu3eConditions *pDC = Mu3eConditions::instance();
    if (pDC->getDB()) {
      calAbs *cal = pDC->getCalibration("pixelalignment_");
      calPixelAlignment *cpa = dynamic_cast<calPixelAlignment*>(cal);
      uint32_t i(99999);
      mu3e::root::alignment_sensors_t entry;
      int cnt(0);
      while (cpa->getNextID(i)) {
        entry.id = cpa->id(i);
        entry.vx = cpa->vx(i);
        // ...
        entry.rowz = cpa->rowz(i);
        entry.colx = cpa->colx(i);
        // ...
        auto& sensor = sensors.emplace(std::piecewise_construct,
                 std::forward_as_tuple(entry.id),
                 std::forward_as_tuple(entry.id,
                        make_float3(entry.vx, entry.vy, entry.vz),
                        make_float3(entry.rowx, entry.rowy, entry.rowz),
                        make_float3(entry.colx, entry.coly, entry.colz)
                       )
                  ).first->second;
  // ...
```

For the current use cases, this rather simple access has been sufficient. It is conceivable that more generic accessors (string based, for instance) could be added and useful.

Apart from providing access to the calibration constants, the concrete DB implementation classes are the only place with the payload BLOB encoding and decoding ("serialization" and "deserialization"). The first eight bytes of the BLOB, its header, contain the value `0xdeadface` and serve as a simple (even visual) cross check that the BLOB encoding is in order. After the header, the BLOB contains numbers (`int`, `unsigned int`, `double`) using 8 bytes for each number. At the moment, no encoding of `long` types is implemented. The number encoding onto the eight bytes is performed with `memcpy` and similar for

the decoding.

```
void calPixelAlignment::calculate(string hash) {
  fMapConstants.clear();
  string spl = fTagIOVPayloadMap[hash].fBLOB;
  std::vector<char> buffer(spl.begin(), spl.end());
  std::vector<char>::iterator ibuffer = buffer.begin();

  long unsigned int header = blob2UnsignedInt(getData(ibuffer));
  cout << "header: " << hex << header << dec << endl;

  while (ibuffer != buffer.end()) {
    constants a;
    a.id = blob2UnsignedInt(getData(ibuffer));
    a.vx = blob2Double(getData(ibuffer));
    // ...
    a.nrow = blob2Int(getData(ibuffer));
    a.ncol = blob2Int(getData(ibuffer));
    // ...
    a.pixelSize = blob2Double(getData(ibuffer));
    fMapConstants.insert(make_pair(a.id, a));
  }

  fMapConstantsIt = fMapConstants.begin();
}
```

In the source code above, `getData(std::vector<char>::iterator)` returns eight bytes and updates the iterator. The BLOB decoding proceeds sequentially through the string, interpreting eight bytes at a time.

Similar knowledge of the BLOB (or the calibration constants of the concrete DB implementation classes) is required for the following functions

- `string makeBLOB()` creates a BLOB using the information stored in `fMapConstants`.
- `string makeBLOB(map<unsigned int, vector<double> >)` creates a BLOB for a map of detector component index and the vector of doubles, which contains (type-cast to double) all numerical information of the `struct constants` for each detector component.
- `map<unsigned int, vector<double> > decodeBLOB(string)` provides the inverse of the previous function.
- `void printBLOB(string, int verbosity = 1)` provides `cout` output of the BLOB.
- `string readCsv(string filename)` read a CSV file containing the numerical values for the constants.

In some concrete DB implementation classes, instead of `readCsv()` a `readJson()` is implemented.

3.3 Database access

The database access is handled through the abstract base class `cdbAbs` with the following minimal interface:

- `init()` provides the database initialization (connecting to the server, if required by the derived class)
- `getPayload(string hash)` retrieves a payload identified through the hash;the concrete payload retrieval is deferred to the derived classes;
- utility functions for reading (global) tags and IOV information in the CBD instance.

The constructor of `cdbAbs` requires a global tag and a URI (filesystem directory or connection URL with authentification for local or remote servers).

At the moment, three different derived classes inherit from `cdbAbs` and are discussed in the following. Switching between the three different implementations is achieved by providing different a specific URI.

```
#include "Mu3eConditions.hh"
#include "cdbAbs.hh"
#include "cdbJSON.hh"
#include "cdbRest.hh"
#include "calAbs.hh"

// ...

opts.add_options()
 ("dbconn", po::value(&dbconn), "DB connection method (json,rest)")
// ...

cdbAbs *pDB(0);
std::string gt = "mcideal";
int dbverbose(10);
if (dbconn == "rest") {
  pDB = new cdbRest(gt,
      "https://eu-central-1.aws.data.mongodb-api.com/app/data-pauzo/endpoint/data/v1/action/",
                  dbverbose);
} else if (dbconn == "json") {
  pDB = new cdbJSON(gt,
      "/psi/home/langenegger/mu3e/mu3eanca/db0/cdb1/json", dbverbose);
} else {
  std::cout << "NO DB connection defined. Using legacy root:alignment/*"
      << std::endl;
}

Mu3eConditions *pDC = Mu3eConditions::instance(gt, pDB);
```

The subsequent code needs no knowledge about the concrete DB implementation—all required calibration data and classes are obtained from `Mu3eConditions`.

3.3.1 Local filesystem with JSON files

The class `cdbJSON` provides an implementation of the CDB with filesystem-based JSON files. As the other two cases, it provides a concrete implementation for

- `vector<string> cdbJSON::readGlobalTags(string gt)`
- `vector<string> cdbJSON::readTags(string gt)`

- `map<string, vector<int» cdbJSON::readIOVs(vector<string> tags)`
- `payload cdbJSON::getPayload(string hash)`

Since the entire database contents is file based, retrieval of the (global) tags and payloads amounts to simply reading files. The JSON files are parsed using code derived from the `bsoncxx` and `mongcxx`. As an example the following listing shows the reading of the JSON file and the preparation of the payload.

```
ifstream INS;
string filename = fURI + "/payloads/" + hash;
INS.open(filename);

std::stringstream buffer;
buffer << INS.rdbuf();
INS.close();

bsoncxx::document::value doc = bsoncxx::from_json(buffer.str());
pl.fComment = string(doc["comment"].get_string().value).c_str();
pl.fHash    = string(doc["hash"].get_string().value).c_str();
pl.fBLOB    = base64_decode(string(doc["BLOB"].get_string().value));
```

3.3.2 Local MongoDB server

The class `cdbMongo` provides an implementation of the CDB with a local MongoDB server. This scheme has the disadvantage that it needs open ports to connect to a MongoDB server which may not be trivial to achieve (given paranoid IT personnel). This implementation does not run on the merlin cluster[6].

The database connection is initiated with `mongocxx::client(uri)`. By convention, the database name is `mu3e`, holding three collections `globaltags`, `iovs`, and `payloads`. The documents in these collections are *exactly* the same JSON-based entities as in the filesystem-based CDB implementation.

3.3.3 REST API cloud based

The class `cdbRest` provides an implementation of the CDB accessing a remote server through a REST API. This scheme uses normal HTTP ports and has no issues with most firewalls. It has been validated to run flawlessly on merlin.

There is no real initialization of the database connection, apart from reading (so far, temporary hack) the authentification key. The connection to the database is achieved through `curl`.

The database setup is identical to the MongoDB implementation, and the stored documents are *exactly* the same JSON-based entities as in the filesystem-based CDB implementation.

It should be noted that the current REST API setup is not at all of production quality. It requires specialized authentification and host address clearing (for interactive access). Its only purpose is to test accessing an outside mongoDB server from within the PSI firewall. That test was successful.

[6]It is not excluded that a dedicated virtual machine hosting a local mongoDB server could be set up at PSI. However, this possiblity has not been pursued for the time being.

3.4 CDB tools

Various tools have been written for specific tasks required for the testing and validation of the CDB framework.

- `cdbInitDB.cc` allows to bootstrap an initial version to a JSON filesystem-based CDB for (currently) two cases: (1) `intrun` reflecting the status (and the then active online chip numbering scheme) of the 2022 integration run and (2) `mcideal` describing the detector in the simulation.
- `cdbPrintPayload.cc` provides a plain-text output of a payload. More output options (*e.g.* CSV or JSON format) could follow.
- `syncCloud.cc` and `syncMongoDB.cc` mirror a JSON filesystem-based CDB to a cloud REST API and local MongoDB server, respectively.
- `insertIovTag` inserts a new IOV for a tag (inserting the runnumber, not necessarily appending).

This small collection of tools is only a start; more would have to follow, *e.g.*. payload visualization and validation, IOV checks, tag management, *etc.*

4 Use case

5 Summary and outlook

References

[1] M. Ritter, et al., *"Belle II Conditions Database"*, Journal of Physics: Conference Series, **1085**(3) 032032, Sep 2018.

[2] S. Di Guida et al., *"The CMS Condition Database System"*, Journal of Physics: Conference Series, **664**(4) 042024, Dec 2015.

[3] D. Barberis, et al., *"Designing a future Conditions Database based on LHC experience"*, Journal of Physics: Conference Series, **664**(4) 042015, dec 2015.

[4] Marco Clemencic, [LHCb], *"A Git-based Conditions Database backend for LHCb"*, EPJ Web Conf., **214** 04037, 2019.

[5] Marko Bracko, Marco Clemencic, Dave Dykstra, Andrea Formica, Giacomo Govi, Michel Jouvin, David Lange, Paul Laycock and Lynn Wood, *"HEP Software Foundation Community White Paper Working Group – Conditions Data"*, 1 2019.

[6] Nik Berger, "Database(s)", Mu3e collaboration meeting March 2022, `https://indico.psi.ch/event/12456/`, 2022.

[7] Mu3e Collaboration, "SpecBook", `https://www.physi.uni-heidelberg.de/Forschung/he/mu3e/restricted/specbook/Mu3eSpecBook.pdf`, 2022.