GlueX-doc-1541-v6

GlueX Calibration/Conditions Database Specification

David Lawrence and Mark Ito

October 4, 2010

Version 0.5

1 Introduction

This document gives specifications and recommendations to be used in the development of the Calibration and Conditions Database (CCDB) for the GlueX detector in Hall-D at Jefferson Lab. The calibration database contains constants that will be determined mostly (if not completely) in the offline environment. The conditions database contains values that are recorded primarily at the time of data acquisition. Both databases will be used in analyzing data from the GlueX detector in the offline environment. As such, it makes sense to use similar technologies for both and maintain both with a common set of tools given the large overlap of their requirements.

The CCDB will consist of the following pieces at a minimum:

- Documentation : Developed at a level that if the original author(s) are unavailable someone else could easily take over maintenance
- Relational Database : Table definitions along with any stored procedures, triggers etc. contained in the database
- Web-based interface : A set of web pages that can be used to browse CCDB values
- C++ library : This should implement a layer that allows the existing JANA API for the CCDB to be used
- Utilities : Command-line utilities useful in actions and maintenance involving the CCDB

It should be noted that an earlier effort was made to develop a calibration database system for GlueX. That effort was documented in a GlueX doc[1]. The current document, while drawing inspiration of that earlier effort, supersedes it.

This design has also been heavily influenced by experience with the CLAS calibration database[2].

2 Database Features

This section describes several features of the database itself. These are meant to ensure certain functionality is available through the presence of specific data and that expected performance requirements are met. The features discussed are:

- Required Columns (columns all tables in the database should have)
- Indexing (primary and secondary index used in selecting constants)
- Namepaths (how constants are identified by detector system)
- Variation (make shallow copies of constants)
- Performance requirements
- Access and Backups

2.1 Required Columns

All entries in the CCDB whether in tables holding the constants themselves or organizational/structural tables, should contain some basic bookkeeping values. These values should contain, but are not limited to, those described in the following subsections.

Author All rows of all tables in the CCDB should have a single author associated with them. For conditions that are entered automatically by the online system during data acquisition, this may indicate the system rather than an actual person.

Creation Time All rows of all tables in the CCDB should have a date/time indicating when the entry was made. This will allow queries of the database where one can specify that they want the values that would have been obtained had the request been made at a specific time in the past. This is critical to allow reproducibility.

2.2 Indexing

A constant set will be identified using a name string, run number, variation string, and optionally an event number.

Constants will be assigned primarily by run number. Each set of constants entered into the CCDB should have a run number range for which they are valid. The range may be as small as a single run number but a facility should exist to specify a set of constants as valid for "all runs"[1]. Current plans anticipate a new run to begin approximately every hour during normal data taking. It is the strong hope that this will be sufficient for indexing since ideally all detector systems will require re-calibration less frequently than once per hour. However, due to past experience and a desire to be prepared for the unanticipated, the CCDB will need to have the ability to record event number ranges for constant sets as well. A facility should exist to specify an event range within the run for which the constants are valid. When an event range is specified, the run range should be required to be exactly 1 run wide. A convention should be implemented to allow the case anticipated to be the most common where a set of constants are marked as valid for "all events in this run"[2].

Event numbers will be reset to start from 1 for each run. The CCDB should be designed in such a way that it will be easy to query it for a list of event number boundaries that can be used to determine when re-reading of calibration constants is required for a specific run.

2.3 Namepaths

The desired set of values will be identified by name. The name string will be free form, but unique across all detector systems. The convention already implemented in the GlueX code base is to use a forward slash(/) notation to specify a hierarchical *namepath*. For example:

```
FDC/driftvelocity/timewalk_parameters
```

This allows implementors of individual detector systems to to specify a hierarchy with as much or little depth as is needed given their complexity.

The "FDC/driftvelocity/timewalk_parameters" parameters may have members identified by either name or position. For example, it may contain 3 values: *"slope"*, *"offset"*, and *"exponent"*. By contrast, a set of constants with a namepath "FDC/CathodeStrips/pedestals" may have 100 values identified simply as "0", "1", "2", "3", ...

[1] This could be done using a simple convention such as RUN_{min}=$\mathrm{RUN}_{max} = 0$.

[2] This could be done using a simple convention such as EVENT_{min}=$\mathrm{EVENT}_{max} = 0$.

2.4 Variations

The CCDB should include a *variation* feature. This will allow one to make a "shallow copy" of a complete set of calibration constants (i.e. one that refers to constants without actually copying them). The variation will be used primarily to create a new set of constants based on another set, but with a few changes. A variation will be identified using a free form string. The primary purpose of a variation will be to allow testing of new constants or alternative sets of constants that don't disturb the main trunk and don't require copying a complete set.

Specifying a variation will be optional for end-users. If no variation is specified, then a *default* variation is used. It is assumed that the *default* variation will be used to hold the best available constants at any point in time. This is a usage detail that can be decided later though after the CCDB comes online and experience with it is gained.

Each *variation* will be stored in the CCDB with information regarding another variation upon which it is based. This basis variation may specify the *default* variation . It should also be possible to specify a basis variation of "none" that indicates no further parent variations should be searched. Having a "none" variation as a parent will allow unrelated sets of constants to be maintained in the CCDB that are completely decoupled from the *default* or main trunk.

Each set of constants in the CCDB will include a variation string that will be used to specify which variation those specific constants are valid for.

It is anticipated that the number of variations may become large and and since many of them will be used privately for testing purposes, it will be difficult to enforce a uniform naming strategy. To help browsing the existing variations, each variation can have one or more text tags associated with it. By default each variation will be tagged with the author's name. Other tags can be introduced when convenient, for example, "production" or "junk".

2.5 Performance

2.5.1 Simultaneous connections

The offline scientific computing farm at JLab currently has on the order of 200 nodes. For the purposes of this document, it is assumed that the same number of nodes will exist when offline processing of GlueX data are present. It is likely that when processing of GlueX detector data begins, many of these nodes will try and access the database simultaneously. The worst case scenario is therefore estimated to be 200 simultaneous accesses to the database as the first batch of jobs starts up. The CCDB should be capable of handling this. It should be noted that at the time of this writing *MySQL* appears to be configured by default to handle 100 connections[3]. Adherence to the 200 connection requirement should

[3] http://dev.mysql.com/doc/refman/5.0/en/too-many-connections.html

be considered as met if the database server is capable of being configured to handle that number.

It should be noted that at this point in time it is also a possibility that the farm makeup will consist of 100 nodes with 100 cores per node. If a single thread process model were employed, that could lead to up to 10,000 simultaneous connection attempts. Handling of that particular scenario will be deferred until such time that it is required.

2.5.2 Access Time

The GlueX detector has roughly 26k channels in it[3]. If there are 2 calibration constants per channel on average, then 52k numbers will need to be retrieved for full reconstruction[4] The values will be grouped in sets that may have as many 10k values in a single set or as little as one value. Access to a set of calibration constants containing a single number should be less than 10 ms using a 1 Gigabit connection to a nearby server. Retrieval of 10k values should occur in less than 1 second under similar conditions.

2.5.3 Replication

The CCDB will need to have replication ability that maintains a duplicate of the database on a separate server. This will be required for maintaining consistency between the offline server and the online server. The online server will fill the conditions part of the database while using the calibrations part of the database for online monitoring. It is likely that some calibrations will be done online so the ability must exist to enter new constants into the calibration part of the online server and have them be replicated on the offline server.

2.6 Access and Backups

The CCDB will most likely be hosted at Jefferson Lab. The CCDB will need to have access by collaborators at remote institutions while still adhering to the cyber security policies of Jefferson Lab.

The CCDB will need to be backed up regularly to ensure important data is not lost. This will most likely be done by the Jefferson Lab IT division using mechanisms already in use for other databases. The only requirement is therefore that the CCDB design be such that it is condoned by the IT division for them to accept responsibility for regular backups.

[4]Channels: CDC(3.5k) + FDC(12.6k) + BCAL(7.1k) + FCAL(2.8k) + TOF(0.2k) = 26.2k. Each channel measures amplitude (offset + gain) and time (1 calibration constant) for a total of 3 calibration constants.

3 JANA API

An important aspect of the CCDB is the implementation of a C++ class that extends the *JCalibration* class in the JANA[4] event reconstruction framework. Code already exists using this API to access constants. Implementing a JCalibration subclass that can access the CCDB will make use of the CCDB instantly available to the Hall-D code base. The JANA API for the CCDB has been documented[5] but enhancements have been made since then to accommodate event-level tagging of calibration constants[6]. These changes are documented in section 3.2.

3.1 Generator Class

Initially, the role of the CCDB has been occupied by a simple set of ASCII formatted files. An API was designed to allow development of the Hall-D reconstruction software using these files until such time that the CCDB comes online. During the development of the CCDB, there exists a transition period during which it is desirable to have both methods accessible in the same executable. Supporting multiple backends simultaneously is done through a traditional object oriented factory mechanism. To supply a backend, one needs to supply 2 classes: 1.) a generator class (*e.g. JCalibrationGeneratorMySQL*) and 2.) a class implementing the backend itself (see section 3.2). The generator class is used to:

1. Determine which backend is appropriate for a specified database
2. Create an object of the appropriate *JCalibration*-based class

```
const char* Description(void);

double CheckOpenable(std::string url, int run, std::string context);

JCalibration* MakeJCalibration(std::string url, int run, std::string context);
```

Figure 1: Methods of the *JCalibrationGenerator* class that must be supplied by the CCDB subclass that inherits from it.

Figure 3.1 lists the methods that must be supplied by a generator class (i.e. one inheriting from *JCalibrationGenerator*). Two of the three methods (*Description()* and *MakeJCalibration()*) are trivial. The *CheckOpenable()* method returns a floating point value between 0 and 1 indicating its ability to open and read from the specified url, run, and variation. A single program may be able to read calibration constants from multiple sources, some of which may come in through plugins and use a technology not built into the initial JANA framework. Having each type of calibration source provide an indication of

its ability to open a specific url allows the framework to decide which should be used based on the external information they provide as opposed to having the decision hardcoded in the framework itself.

The initial implementation of the MySQL-based CCDB may have a *CheckOpenable()* method that simply returns 0.1 if the url begins with the string "mysql://" and returns 0 otherwise.

The *url* is a free-form string indicating the protocol and location of the CCDB data. For example, this could have a form:

file:///group/halld/calib

or

mysql://halld_user@halld_db.jlab.org

3.2 Implementation Class

The class that actually implements the communication with the CCDB database is a subclass of the *JCalibration* base class. The methods of *JCalibration* that must be implemented are shown in figure 3.2. These define the API that provides the following functionality:

- Retrieve calibration constants from CCDB (*GetCalib()*)
- Insert calibration constants into CCDB (*PutCalib()*)
- Discovery of available constants (*GetListOfNamepaths()*)
- Identification of event-level boundaries (*RetrieveEventBoundaries()*)

Values are retrieved or inserted into the CCDB using STL container classes. 1-D arrays are communicated using a *map<string, string>* container with the column name being the key[5] and the value being the value[6]. The key will hold the actual CCDB column name, if the values are stored in that way, or the value position (e.g. "0", "1", "2",...). See section 4.

For 2-D tables, an STL container of type *vector<map<string, string> >* is used. Each element of the vector is one row of the table. Note that this keeps the column names with every row which is redundant, but also provides some convenience.

The CCDB should also implement the *RetrieveEventBoundaries(void)* method. The method will obtain a list of event numbers from the CCDB for the given run number/variation for which an event-level boundary exists for any *namepath* contained therein.

[5] map<string,string>::first

[6] map<string, string>::second

```
// Returns "false" on success and "true" on error
bool GetCalib(string namepath, map<string, string> &svals, int event_number=0);

bool GetCalib(string namepath, vector< map<string, string> > &svals, int event_number=0);

bool PutCalib(string namepath, int run_min, int run_max, int event_min, int event_max
            , string &author, map<string, string> &svals, string comment="");

bool PutCalib(string namepath, int run_min, int run_max, int event_min, int event_max
            , string &author, vector< map<string, string> > &svals, string comment="");

void GetListOfNamepaths(vector<string> &namepaths);

void RetrieveEventBoundaries(void);
```

Figure 2: Methods of the *JCalibration* class that must be supplied by the CCDB subclass that inherits from it. Note that the Run Number and Variation are supplied as member data through the *JCalibration* base class.

Having this list of boundaries will allow the JANA framework to make the proper callbacks to re-read calibration constants whenever an event-level boundary is crossed. The event boundaries should be stored in the *vector<int> event_boundaries* member of the JCalibration base class.

4 Table Definitions

The database tables express objects and relations among components in the problem space being addressed: that of developing, applying, and modifying calibration constants for the various detector components.

There are several times associated with any set of data. One of them is the time that the data was taken. The others are the various times at which they were calibrated (more precisely times at which a declaration is made about which constants should be used for a given set of data). The latter may happen many times. Only this latter meaning is relevant for this database discussion. The time that the data was taken is irrelevant. In this sense data may be "calibrated" (have constants assigned) even before they are taken.

The tables are described below and their relations are shown graphically in Fig. 3. The tables themselves are described in tables 1-8.

The central table is created as a join of the constCorr, runRange, constantSet, constantType, and variation tables. As such, the following table can be formed unambiguously for each row of the constCorr table:

```
constantType.id
runRange.runMin
runRange.runMax
```

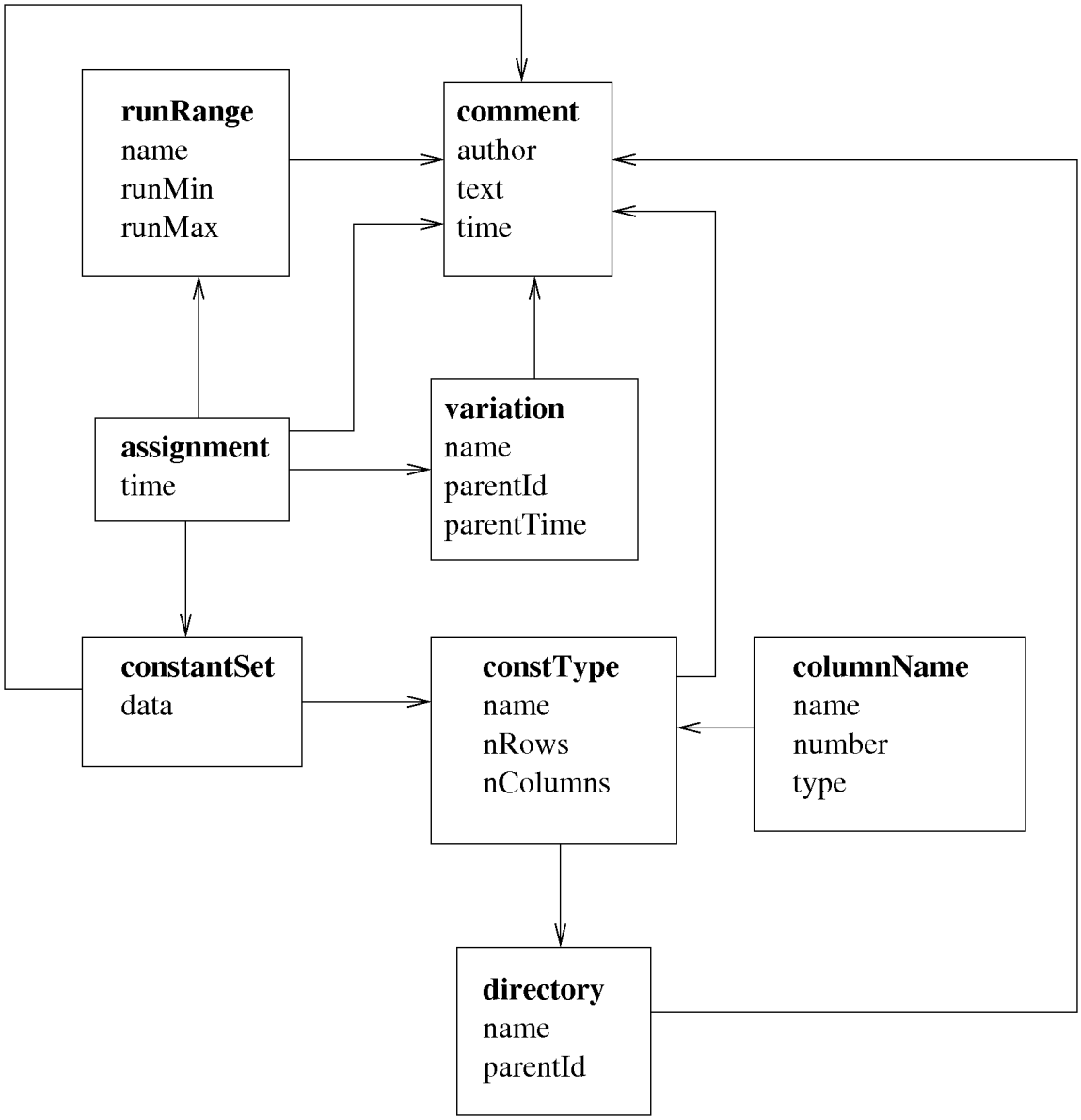

Figure 3: Relationships between tables of the calibDB. Arrows point to the the table being referenced. The table at the source of the arrow thus contains foreign keys identifying a row in the referenced table.

Table 1: Definition of *constCorr* table.

Column Name	**Column Type**
id	int
time	timestamp
runRangeId	int
constantSetId	int
variationId	int
author	varchar
time	timestamp
commentId	int

Table 2: Definition of *runRange* table.

Column Name	**Column Type**
id	int
name	varchar
runMin	int
runMax	int
author	varchar
time	timestamp
commentId	int

Table 3: Definition of *constSet* table.

Column Name	**Column Type**
id	int
constTypeId	int
data	blob
author	varchar
time	timestamp
commentId	int

Table 4: Definition of *variation* table.

Column Name	**Column Type**
id	int
name	varchar
parentId	int
parentTime	datetime
author	varchar
time	timestamp
commentId	int

Table 5: Definition of *constType* table.

Column Name	**Column Type**
id	int
name	varchar
rows	int
columns	int
directoryId	int
author	varchar
time	timestamp
commentId	int

Table 6: Definition of *directory* table.

Column Name	**Column Type**
id	int
name	varchar
parentId	int
author	varchar
time	timestamp
commentId	int

Table 7: Definition of *columnName* table.

Column Name	**Column Type**
constTypeId	int
name	varchar
number	int
type	enum
author	varchar
time	timestamp
commentId	int

Table 8: Definition of *eventRange* table.

Column Name	**Column Type**
id	int
run	int
eventMin	int
eventMax	int
constSetId	int
time	timestamp
commentId	int

```
constSet.array
```

where constantType.id is the primary index of the constantType table. For a given constantType.id, there is an run range of applicability and an associated set of constants. This minimal set of tables would suffice, but has several disadvantages. Only one set of calibration constants for a given run can exist at one time. The changes made to the constants are lost and thus reconstruction done in the past with old constants cannot be reproduced. The run ranges must be non-overlapping or ambiguities about which constants should be supplied would arise.

For the next step we add the time of constant set assignment:

```
constantType.id
runRange.runMin
runRange.runMax
constSet.array
constantCorr.time
```

Now the run range data can specify overlapping ranges. In case of overlap the constants that go with the most recent time are taken. If we also never delete old constant assignments, this practice also provides a history mechanism: constant set assignments that were valid at any given time can be reproduced by only accepting rows that existed before that time. In this way the state of the table at previous times is preserved and can be accessed.

Finally, we add another column marking a variation of the assignments:

```
constantType.id
runRange.runMin
runRange.runMax
constSet.array
constantCorr.time
variation.id
```

Now constants sets are assigned only for particular variations, where the variations are in an hierarchy expressed in the variations table. It is anticipated that each variation will only deal with a relatively small number of constant set assignments. If runs or constant types are requested of a variation that are not assigned within that variation, then the parent variation is queried, recursively, until a valid constant set assignment is found.

4.1 How to get constTypeId

The constType.id is the primary key for the constType table. When the user requests a particular set of constants, the request will specify the type of constants required, with the type specification coming in the form a full "path" specifying the name(s) of one or more nested directories as well as the name of the constant type, much like a file in a directory

hierarchy. By using the directory table the record for a particular constant type can be found and thus constType.id can be retrieved. Footnote: in practice a join will be used that uses constType.id only implicitly, but this discussion illustrates the concept.

4.2 Event Ranges

If event-level assignment of constants (see Sec. 2.2) is necessary. Then the eventRange table is used. An assignment of constats on the event level is indicated by constSetId = NULL in the assignment table. In that case the assignment is done by consulting the join of the eventRange and constType table. If no suitable event range is found, then the event range assignment for eventMin = eventMax = 0 is used. Note that the same history mechanism used for run range assignments works in the event range case as well.

5 Recommendations

The software technologies chosen to implement various aspects of the CCDB should give strong consideration to the eventual deployment targets. Namely, if the CCDB related software is something that one can expect to be used by collaborators at remote institutions then commercial software is to be avoided wherever possible and preference given to open source packages. It is also recommended that the number of different packages be minimized in order to reduce the burden on new installations that will use the CCDB related software.

For the reasons described above, it is recommended that *MySQL* be used as the database backend. Besides being open source, widely used and well documented, it has been used extensively at JLab and so local expertise and experience exist.

Similarly, PHP should be considered for the web interface to the CCDB.

Some command line tools based on JANA already exist and so will be instantly useable once the appropriate JCalibration subclass is made. These tools will provide some basic copying from flat ascii files to and from the database. They will, however, be insufficient for the range of management tools that will be required. While these tools may be written in almost any language, use of a scripting language such as Python or Perl or a language such as Java that produces portable binaries is recommended. The Python option seems particularly attractive at the moment since it is currently expected that Python will play a large role in the online environment where some access to the CCDB via scripts will be required.

References

[1] N. Kolev, "Hall D Calibration Database Table Design and Interface," GlueX Doc-672.

[2] Harut Agavyan, Mark Ito, Greg Riccardi, Riad Suleiman, "The CLAS Calibration Database," CLAS-NOTE 2001-003.

[3] Fernando Barbosa, "Summary of Hall D Detector Systems," GlueX-Doc 747.

[4] D. Lawrence, "Multi-threaded event reconstruction with JANA," Journal of Physics: Conference Series, **119**, 4, pp. 042018 (2008).

[5] David Lawrence, "The JANA calibrations and conditions database API," Journal of Physics: Conference Series, **219**, 4, pp. 042011 (2009).

[6] http://www.jlab.org/JANA/downloads.php