Hall D Conditions Database Schema

Elliott Wolin and David Lawrence

5-Dec-2011

In this note we propose a schema for the Hall D conditions database. This proposal is based on Mark Ito's original presentation at the 13-Apr-2011 Hall D Online meeting, as well as on discussions with Mark, Dmitry Romanov, Vardan Gyurjyan (DAQ group), Bobby Lawrence (MIS group), members of the CLAS12 software group and other Hall D members.

General Considerations

The conditions database must satisfy numerous requirements, some of which are only partially understood at this point. Requirements include:

1. Entries will be made from scripts (Python, Shell, perhaps Perl) and programs (Java, C++) run as part of Hall D Online systems.
2. Most entries will be made within the first 10-20 seconds of the start of a run, where we expect 10-20 processes will make 100-200 entries with a maximum aggregate size of less than 1 Mbyte. Additional entries will be made at the end of a run or during the run. There are scenarios where we may exceed 1 MByte, but these don't appear likely at this point.
3. Entry time is not critical, as most if not all entries will be made by background processes.
4. Conditions data must be easily available to offline reconstruction programs via an API as similar as practical to the one developed for the offline calibration database.
5. There is no need to duplicate the functionality of the EPICS archiver database, although for convenience some data may appear in both (e.g. solenoid current setpoint at the start of a run).
6. Other EPICS archiver data needed by offline analysis programs will be extracted from the EPICS database, processed and entered into the offline calibration database.
7. Flexibility is very important since it is difficult to anticipate the number and kinds of conditions data that will need to be stored, and we further expect this will change as the experiment matures.

8. We should use MySQL, same as the offline calibration database, but should implement everything in as a database-independent manner as possible in case we ever need to switch to another database system.

Schema Design

These general considerations lead to a number of design choices.

1. Use a "normalized" table structure for the bulk of the data, similar to what is done in the offline calibration database. That is, rather than using a schema composed of a series of multi-column tables ("un-normalized"), instead store the bulk of the data in a single table that includes a data type ID, with each ID corresponding to a column in an un-normalized schema. In this way one can easily add new ID's without affecting anything else. Special ("cross-tab") query techniques can be used to get data in an un-normalized form.
2. Values will be stored in large capacity text fields to retain maximum flexibility. These can hold single numerical values, short strings, large arrays of numbers or long XML strings.
3. A data type ID table will list and document all data types stored in the data table.
4. If needed, data types can be of a "foreignKey" type, i.e. one whose value corresponds to a primary key of another table in the conditions database.

These foreign key types might be useful in the following scenario. Imagine you need to store configuration data that includes many related numbers. One possibility is to pack the numbers into a string and store the string in the text field of the main data table. Some concerns are: 1) do you store this for every run even though the numbers may not change often, 2) if not, how do you label them as a set so you know that they are the same set for multiple runs, and 3) how do you perform database queries on the individual numbers?

Each of these concerns can be handled in a variety of ways, and we propose the following. Instead of storing the set of numbers in the main data table, store them in an auxiliary table indexed by a "set number", then just store the set number in the main data table. As with the main data table, the auxiliary table might be normalized to allow for flexibility and expansion, but this is not required, and may not be desirable in some cases.

Proposed Schema

Our proposal for the main data table is:

Column Name	Column Type	Comment
run	int	Run number corresponding to this entry
typeID	int	ID found in the data type table
value	text	Flexible, can hold just about anything
dataTime	dateTime	Time corresponding to this entry
entryTime	timestamp	Time this entry was added to the database

An additional possibility is to add a column containing a checksum of the value field to allow for fast comparisons between the value field and text strings.

Our proposal for the data type table is:

Column Name	Column Type	Comment
typeID	int	ID for this data type
name	varchar(64)	Name of this data type
type	varchar(64)	Type of this entry (int, float, XML, foreignKey, etc.)
specification	text	XML specification of this type, if needed
description	text	Description/comment
creationDate	timestamp	Time this type was created

An auxiliary table might be:

Column Name	Column Type	Comment
auxKey	int	Key or set number stored in the value field of the main data table
index1	int	Some sort of identifying index
…	…	More indices, if needed
value	int or float or …	The value, type as appropriate
…	Int or float or …	Additional values, if needed

where the auxiliary table name is stored in the data type ID table in the specification column (see below).

Offline Analysis API

The offline analysis software is written in C++ and so a C++ compatible library is required. The API will be consistent with the existing offline framework, which implements access to the calibration database. Specifically, the library should include a C++ class that can be used to

access conditions for a specific run with the run number maintained in the class itself, along with any other relevant indexing information. This will allow the various offline packages to request values only by name and event number without having to explicitly specify information such as run number, server, etc.. The framework (JANA) will be responsible for creating a new instance of the conditions database interface object (CDBI) for each run number as it is needed.

The data obtained from the conditions database can come in the form of single values, 1-D arrays of values, or 2-D tables of values. For the arrays and tables, values may be indexed by position or by a name (key) depending on what is appropriate given the nature of the data. The following table lists the STL data types for these five data format possibilities. The value "T" represents the fundamental data type (double, float, int, string, …) that the data itself will be converted to. The CDBI class will implement templates so all fundamental data types will be implemented with only a few methods.

Data Types	
Single	T
Array (position indexed)	vector<T>
Array (name indexed)	map<name,T>
Table (position indexed)	vector<vector<T> >
Table (name indexed)	vector<map<name,T> >

The most commonly expected use case for the conditions database will be to ask for a named set of values appropriate for the current run number and event number being analyzed. For this case, a set of five templated *Get()* methods corresponding the five Data Types listed above will be supplied by the CDBI class. In addition, a set of five *GetAll()* templated methods will be supplied to allow users easy access to the complete list of values for the current run number in the case of multiple entries per event. The format of these methods is given here where "TT" represents one of the data types listed above:

<template T> int Get(const string name, TT &vals) const;

<template T> int GetAll(const string name, multimap<time_t t, TT> &vals) const

Database Usage Examples

One simple example is recording the start time of a run. Assume data type 1 is named "runStartTime" and is of type "posixTime." No specification is needed. Entries in the main data

table would consist of a run number, typeID=1 and a run start time in the value field (here the field dataTime is redundant). Similar examples include the target ladder position, solenoid current setpoint, names of operators on shift, a comment for the run, the binary program stored in an MLU unit in hex, etc.

A simple use of auxiliary tables might be as follows. Assume configuration of some board involves 20 numbers, and that you want to perform queries on the individual numbers (e.g. find all board configurations where the third number is less than 17 and the fifth is greater than the ninth). The corresponding data type (e.g. number 8) might be named "boardConfigParams" with type "foreignKey." The specification field might be set to something like:

<auxTableDef name="boardConfigTable">

Here the auxiliary table named "boardConfigTable" might have three columns:

Column Name	**Column Type**	**Comment**
setNumber	int	Used as foreignKey in main data table
paramNumber	int	Parameter number in board
valCol	int or float or …	The value, type as appropriate

The data table would include an entry for each run having typeId=8 and value equal to the set number used for this run. If the set number is 23 the auxiliary table would include 20 entries with setNumber=23, one for each board parameter. Other auxiliary table definitions are possible as are alternate schemes for the specification field. For example, if you were absolutely sure the number of board parameters will never change one might consider creating a boardConfigTable with 20 value columns, one for each board configuration parameter.

Summary

Based on current expectations the schema presented above is adequate to store all configuration data not stored by the configuration database, EPICS archiver or offline calibration database. It is extremely flexible and expandable, and can accommodate the use of auxiliary tables when normalization of the data is not desired.