Data Storage Schema for the LHC Super-Table

Elliott McCrory, for the LAFS Team

16 June 2010

We propose a storage schema for the Super-Table information for the LHC.

# Assumptions

For the prototype, we will implement the storage of the super-table with SDDS files. These files initially will be part of the Java code package, under the control of SVN.

Mario has implemented database tables for the LHC Super-Table, but they are not configured to this schema.

## Definitions

**Element** refers to a basic persistent storage unit. For the SDDS implementation, this is an SDDS File. For a database implementation, this is a database table, or a column in a table.

# Storage Elements Needed

The following persistent storage elements are needed.

1. The list of fills that have been recorded, called FillNumbers.
2. The column headers, called ColumnHeaders.
3. The super-table information, called SuperTable. This element breaks into the following sub-elements, one per fill:
	1. The super-table values for that fill, one per column of the Super-Table. This element is called TableValues.
	2. The units for the cells. This element is called Units.
	3. The list of the algorithms used for the calculation of each cell. This element is called Algorithms.

Each of these elements are now specified.

## FillNumbers

This element has one field called FillNumber. Reading this element returns an Integer Array of values.

The name of the SDDS file is FillNumbers.sdds. The suggested name for the database table is FillNumbers.

## ColumnHeaders

This element has one field called ColumnHeader. Reading this element returns a sorted String Array of the column headers. The index into this array matches the index into the array of values in the cells of the super-table.

The name of the SDDS file will be ColumnHeaders.sdds. The suggested name for the database table is ColumnHeaders.

## Fills

This element has the following fields:

* The fill number
* The column number, which is an index into the ColumnHeaders element
* The value for this cell. This is the TableValues element.
* The Units for this cell. This is the Units element.
* The Algorithm for this cell. This is the Algorithms element.

The fill number plus the column number creates a unique index into this element.

### TableValues

This is a series of elements that is stored as individual elements, one per fill. Reading this element returns a String Array of the values in each column. The value in the first column is at array index=0 (but we refer to this as “Column 1”).

The name of the SDDS files will be Fill\_000000.sdds, where the six-digit “000000” is replaced with the zero-padded value of the fill.

### Units

This is a series of elements that is stored as individual elements, one per fill. Reading this element returns a String Array of the Units for the values in the cells. The index into this array matches the index into the array of values in the cells of the super-table.

This element is stored in the same archive as the TableValues, with the selector (column) name “Units”.

### Algorithms

This is a series of elements that is stored as individual elements, one per fill. Reading this element returns a String Array of the algorithms for the values in the cells. The index into this array matches the index into the array of values in the cells of the super-table.

This element is stored in the same archive as the TableValues, with the selector (column) name “Algorithms”.

## Summary of Archiving Elements

In summary, the following archiving elements are defined for the SDDS implementation:

|  |  |  |
| --- | --- | --- |
| **Archive Element** | **SDDS File Name** | **Which contains:** |
| FillNumbers | Fills.sdds | All the fill numbers that have super-table data |
| ColumnHeaders | ColumnHeaders.sdds | The descriptive names of the super-table columns |
| TableValues | Fill\_000000.sdds | The actual super-table values (may NOT be null) |
| Units | The units for the values in the super-table (may be null) |
| Algorithms | The descriptive algorithm for the value in the super-table (man NOT be null) |

For the database implementation, we recommend this implementation:

|  |  |  |
| --- | --- | --- |
| **Archive Element** | **Database Table** | **Which contains:** |
| FillNumbers | Fills | All the fill numbers that have super-table data |
| ColumnHeaders | ColumnHeaders | The descriptive names of the super-table columns |
| TableValues | SuperTable | The actual super-table values (may NOT be null) |
| Units | The units for the values in the super-table (may be null) |
| Algorithms | The descriptive algorithm for the value in the super-table (may NOT be null) |

The specific database schema is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Database table** | **Field** | **Type** | **Null?** | **Index** |
| **Fills** | FullNumber | INTEGER | NO | YES |
| **ColumnHeaders** | ColumnNumber | INTEGER | NO | YES |
| ColumnHeader | VARCHAR | NO | NO |
| **SuperTable** | FillNumber | INTEGER | NO | YES\* |
| ColumnNumber | INTEGER | NO | YES\* |
| Value | VARCHAR | NO | NO |
| Units | VARCHAR | YES | NO |
| Algorithm | CLOB | NO | NO |

\* The FillNumber plus the ColumnNumber are unique in the table and together make a compound index.

# Java Class Definitions

To facilitate the reading and writing of these storage elements (SDDS files or database tables), the following Java classes will be created:

1. Write the storage elements for a specified fill number, given the data for the fill. This class will be called STSave.
2. Read the known fill numbers. This class will be called STFills.
3. Read the defined columns in the super-table. This class will be called STColumns.
4. Read the storage elements for a specified fill number. The client will be able to get the data for the fill from this class. This class will be called STRead.

Each of these classes will be implemented as an interface, and implemented into a concrete class called *Xxx*Impl.

## Public Interface STSave

**public** **interface** STSave {

 **public** **void** write(**int** fillNumber) **throws** IOException;

 **public** **void** setValue(**String** column, String theValue) **throws** IllegalArgumentException;

 **public** **void** setUnits(**String** column, String theUnits);

 **public** **void** setAlgorithm(**String** column, String theAlgorithm) **throws** IllegalArgumentException;

}

Of course, specifying that a method throws IllegalArgumentException is not strictly necessary. It is included here to force the client to understand that these methods cannot have a null argument.

It is expected that the general flow of the program that writes the archive will be something like this:

 STSave archiver = STFactory.*getSave*();

 archiver.initialize(fillNumber);

 **for** (**int** c = 0; c<numColumns; c++) {

 STCell cell = getCellClass(c);

 String value = cell.getValue();

 String units = cell.getUnits();

 String algorithm = cell.getAlgorithm();

 String column = columnNames[c];

 archiver.setValue(column, value);

 archiver.setUnits(column, units);

 archiver.setAlgorithm(column, algorithm);

 }

 archiver.write();

## Public Interface STRead

**public** **interface** STRead {

 **public** **void** read(**int** fillNumber) **throws** IOException;

 **public** **int** getNumColumns();

 **public** String getValue(**int** colNum);

 **public** String getUnits(**int** colNum);

 **public** String getAlgorithm(**int** colNum);

}

Reading an archive file for the super-table data is done through this interface.

## Public Interface STFills

**public** **interface** STFills {

 **public** **int**[] getFills();

}

## Public Interface STColumns

**public** **interface** STColumns {

 **public** String[] getHeaders();

}

## Public Interface STUnits

**public** **interface** STUnits {

 **public** String[] getUnits();

}

It is likely that STFills, STColumns, STUnits and STRead will be implemented in a single class.

## Other Classes

There needs to be a class to write the column descriptions (initially and when they change) and one to write the fill numbers (initially).

# Issues

This schema assumes that the order of the columns is fixed. This is not a great assumption, as columns will be added, removed or re-arranged in the Super-Table database. If necessary, there can be a redirection in the column specification. For example, one can create an invariant mnemonic for each column that indexes into a new table to give the actual column number. (This mnemonic could be an eight-character string, for example.) The Super-Tables cells are then stored by fill number plus column mnemonic.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Database table** | **Field** | **Type** | **Null?** | **Index** |
| **Fills** | FullNumber | INTEGER | NO | YES |
| **ColumnHeaders** | ColumnMnemonic | CHAR(8) | NO | YES |
| ColumnNumber | Integer | NO | NO |
| ColumnHeader | VARCHAR | NO | NO |
| **SuperTable** | FillNumber | INTEGER | NO | YES\* |
| ColumnMnemonic | CHAR(8) | NO | YES\* |
| Value | VARCHAR | NO | NO |
| Units | VARCHAR | YES | NO |
| Algorithm | CLOB | NO | NO |

It is assumed that the generation of the super-table entry for a new fill will be initiated by hand. Ultimately, this should be initiated automatically.