SDA Time Abstraction and Tagging Raw Datalogger (Timber) data.

```
...
The "time abstraction" is clearly something we
will need in LHC. I have always seen this as a
layer on top of the "normally" UTC-time-stamped
data, that can be used as a filter to use another
parameter than "time" to get to the data. Is this
what they propose, i.e. SDA generates this
filtering capacity, based on parameters such as
LHC-shot-number (#37), LHC-mode  ("Filling"), Set
(2nd injection), Collection (Intensity-devices) ?
OR... Is the raw data stamped with this kind of
information. In any case, LHC Logging is designed
since 2001, implemented since 2003 and
operationally used since 2004. Work would we
useful on this  "time abstraction" layer, but
without turning the existing system upside down.


Ronny Billen, from e-mail letter “Response from
key CERN people on SDA”
```

Filtering is one of the SDA features. In Fermilab currently it is implemented in OSDA API and in Data Logger Viewers (original D44, Java Data Logger Viewer family). API lies kind of “aside” of mainstream tools – you need to write some programs to get this filtering. In tools it is not known by many people – so, sigles use it. Despite that filtering is one of the most valuable feature of OSDA.

The idea of direct “filtering” or “tagging” data worth discussing. It may be implemented easily in the SDA with both “postprocessing” approach and “on-line” approach. “On-line” may be even easier. Originally SDA is somewhat like “turning the existing system upside down” - in Fermilab historically it is 2 completely independent systems – different dabase servers, different concepts etc. I still consider that “independant SDA logging” necessary, because it is convenient for building summary tables and extremly valuable for debugging. Big set of tools is based on the fact that exactly one “right for this situation” value corresponds for given device in “Shot#37 – Filling - Injection 2”. This value should be collected under “correct” circumstances to describe what is happening in “Shot#37 – Filling - Injection 2”. In our words – it should be sampled on the right event. I also see potential usage for big pieces of data, impossible to store periodically, collected on the right event and saved in independent SDA DB or DB table.

Back to “filtering”. How it may be implemented? Every shot, case, set, collection are armed, started and disarmed on events. Lets imaging the table with columns “Owner” (in our case owners are Collider, PbarTransfer, Recycler), “Shot” (“Fill” in CERN terms), “Case”, “Set”, “Collection”, “Overstore” (roughly – was it successful), “Start_Time_Stamp”, “Finish_Time_Stamp”. With all the columns indexed.

With such a simple scheme one can create the SQL queries taking in consideraton the desired mode (owner, shot, case, set) and select data for this conditions from datalogger. Vice versa it is possible to build a hierarchical tree of time intervals, that looks like this:

[shot XXX, start, stop [[case YYY, start, stop [[set 1, start, stop]... more sets]]...more cases]]

On such tree it is easy to define some kind of time intervals arithmetic and it is easy to build the list of time intervals, that corresponds, for example, to “first sets of HEP for ColliderShot for the last year”, after that you you may / can select data for this periods of time and do something with them (plot them with overlapping regarding the start of HEP, process them in user command etc). You can also easily show “color” your raw datalogger data with tags like “Shot#37 – Filling - Injection 2”. It is also possible to write somewhat like SDA Viewer over such a scheme.

Described schemes were tested and reference implementation is provided. Java class gov.fnal.sda.db.models.AlaCERNModel, which can be found in FNAL Controls CVS, implements such a tree and such a filtering as desribed. It builds some simple set of tables, fills them with random data, builds random SDA trees (with some “predefined” structure) and makes all of the described “filtering”, “coloring” and “time abstraction operations”. This class may be the base and the sandbox for the real system.

To implement a complete system (Data Collection) for CERN that has these features SDA needs some REALLY MINIMAL changes in just one plugin - implementation of gov.fnal.sda.db.DaqDataWriter, currently gov.fnal.sda.db.xmldb.XmlDaqDataWriter. Roughly - 6 SQL queries ...

The Model.

The model includes 5 tables: **raw_datalogger, sda, device_names, owner_names, case_names**. Last three tables allows for naming devices, owners, cases respectivly. Those tables are not interesting, because substitutions of deviceId->name, ownerId -> name and ownerId, caseId -> can be done at other level, not necessarily in SQL on in model at all. All the tables are created in the method called **clearModel**.

raw_datalogger is a table that maps **unique(device_index, timestamp) -> double_value**. That is an obvious (and I hope “accurate enough”) abstracttion for CERN datalogging scheme. As far as I know CERN team does not have complicated bucket - based distributed dataloggers scheme trading price of Db servers, reliability, performance, and size of logged data for simplicity of SQL-based programming API and comfort of Oracle support.

sda table can be described as “SDA timestamps and nothing else”. It has many fields describing all the possible SDA structure. So, for us it is first of all mapping

indexed(ownerId, shot, case, set), unique(shot_index, collection_index), valid -> start_ts, finish_ts

The contens of the **raw_datalogger** and **sda tables** can be dumped by methods

dumpRawDataLogger and **dumpSDA**. In these dumps names from **device_names, owner_names, case_names** are used for comfort analisys.

In the model we used HSQLDB in memory for speed and simplicity. To allow easy switch to other relational DB servers Db connection was encapsulated into DbServer internal class. Data for **raw_datalogger** and **sda** tables was randomly generated ensuring that all the general assumptions about SDA data are true: some of the cases overlapps, there are several shots for different owners going on simultaneously, some of the case are multi-set, some are not, there are “non-cocurrent” cases monopolized SDA for given owner etc. The model is filled with random data in the method **makeUpData**.

Goal.

The goal was to test the approach, try to filter timestamped device readings using SQL statements, play with first version of Time Abstraction API for SDA, described in the next chapter.

Hierarchical Time Abstraction for the SDA (from Javadoc).

Class **SdaTimeInterval** from package **gov.fnal.sda.osda** represented the time abstraction has a "logical level" according to hierarchy:

general (0) -> owner (1) -> shot (2) -> case (3) -> set (4)

"Logical Path" information is held in the logicCoords int array: logicCoords[0] -> owner, logicCoords[1] -> shot, logicCoords[2] -> case, logicCoords[3] -> set. Each of them may be -1 - uninitialized. So, set node has all of its logicCoords >= 0, case node has logicCoords[3] == -1 etc.

The rule : **if logicCoords[i] == -1 and i < j then logicCoords[j] == -1;**

Besides logicCoords there are uniqueCoords, that may serves as database key (recall overstored shots and collections). Flag valid == not overstored. uniqueCoords has similar meaning and rules as logicCoords. For convenience each TimeInterval should have ownerName and caseName. Important: **Each SdaTimeInterval should have its logicCoords, uniqueCoords and names unmodifiable**.

Each SdaTimeInterval may have children of higher level.

SdaTimeInterval define some obvious getters : getOwner, getOwnerName getShot, getShotName etc.

SdaTimeInterval define some obvious time interval arithmetic : includes, overlaps, insideOf, before, after, includes for long timestamp

SdaTimeIterval also have several not trivial methods:

public SdaTimeInterval refine(int[]refinement) - can be used, for example for selecting all first sets of HEP for all shots for extended period of time, for example:

```
SdaTimeInterval year = XXX.buildTimeInterval(Today - OneYear, Today);
```

```
List < SdaTimeInterval > allFirstHEPSets = year.refine( new int[1, -1, 14, 1] ).getLeafs();
```

public SdaTimeInterval getRelevant(long ts) - builds new time interval, where only shots, cases and sets, which includes this particular timestamps are left. Can be used for "coloring" of datalogger data, for example:

```
SdaTimeInterval shot = XXX.buildTimeIntervalForShot(new int[]{1,5008,-1,-1});
List < Pair < Long, Double > > protonIntensityOfBunch1
            = datalogger.read("C:FBIPNG[1]", shot.getStart(), shot.getStop() );
for( Pair < Long, Double > p : protonIntensityOfBunch1 ){
    List < SdaTimeInterval > collections = shot.getRelevant( p.first() ).getLeafs();
    print( new Date( p.first() ) ); print( "\t" ); print( p.second() ); print( "\t" );
    print( "System state: \t" );
    for( SdaTimeInterval coll : collections )
            { print( coll.getCaseName() ); print( ":" ); print( coll.getSet() ); }
    println();
}
```

public final boolean insertAccordingToLogic(SdaTimeInterval interval) includes child if it can be included. It includes it in appropriate branch etc.

public boolean differentBranch(SdaTimeInterval interval) determines that this and interval are on the same branch of (SDA Viewer) tree

public String toXml() - obvious, needed for serialization. Serialization together with something like

public static SdaTimeInterval fromXml(String xml) is not implemented yet.

Results.

Advantages:

1. It is obvious that filling such a **sda** table ("SDA timestamps and nothing else") requires minimal changes in just one implementation of interface gov.fnal.sda.db.DaqDataWriter, currently gov.fnal.sda.db.xmldb.XmlDaqDataWriter. Roughly - 6 SQL queries. So, it comes for free with "normal" system.
2. SdaTimeInterval can be filled from XML database as well. May be that "offspring" time abstraction API will require another plugin for reading.
3. Filtering of the device readings on the SQL level using SQL WHERE condition and JOINs is possible and have decent performance. I did not measure it, because it has no sense now – only in working Db server situation.

4. Hierarchical Time Abstraction API is decent in terms of convenience and allows for all necessary timestamp / time intervals manipulations.
5. Hierarchical Time Abstraction API allows for writing analogs of SDA Viewer.
6. It is lightweigth and does not include any of the “not standart” XML databases.
7. The amount of configuration needed to bring system up and running is minimal – minimal attention from CERN people required.

Disadvantages (in case there is no “normal” SDA implementation):

1. System cannot guarantee that something will be read / stored for some particular collection. Simulation shows such a “holes” regulary.
2. Ability to store “big chunks of data” collected in right amount of time (“sampled on the right event”) is gone.
3. System does not works as a backup for usual datalogger.