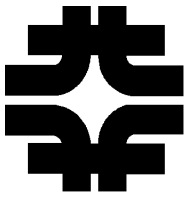


Fermilab

Accelerator Division/LHC Accelerator-Fermilab Software (LAFS)

Friday, February 16, 2007

Timofei Bolshakov
Elliott McCrory
Dennis Nicklaus
Jerry Cai

We present here an implementation to add functionality to the existing Timber datalogger database in order to provide SDA-like functionality. In particular, the ability to address data in the datalogger using the temporal abstractions of SDA is created. This proposal has been prototyped at Fermilab. Database tables and Java interfaces to these tables are presented.

All of the Java classes are implemented in the package **gov.fnal.sda.db.models** as the class **AlaCERNModel.**

The Database Model

A database model has been created to represent the data in the Timber database as if it were SDA data, that is, the data are indexed through the time abstractions defined by SDA: Shot, Case and Set. The model consists of these five database tables:

- **raw_datalogger**
- **sda**
- **device_names**
- **owner_names**
- **case_names**

The last three tables allow for the naming of the devices, the owners and the cases, respectively.

All the tables are created in the Java method called **clearModel**.

It is possible to implement this translation in other ways since substitutions of deviceId → name, ownerId → name and ownerId, caseId → name can be done at a different level, not necessarily in the SQL or in the model at all.

raw_datalogger is a table that maps **unique(device_index, timestamp) → double_value**. That is an abstraction for the CERN datalogging scheme that should be accurate enough for these purposes.

This assumes a straight-forward UTC time indexed datalogger (the Fermilab "Lumberjack" datalogger is not).

The **sda** table contains only SDA timestamps. It has many fields describing all of the possible SDA structure—it is first of all mappings,

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

The contents of the **raw_datalogger** and the **sda** tables can be dumped by the methods **dumpRawDataLogger** and **dumpSDA**. In these dumps, names from **device_names, owner_names, case_names** are derived and presented.

We have used HSQLDB for speed and simplicity. To allow an easy switch to other relational DB servers, the database connection is encapsulated into the internal Java class called **DbServer**.

It is necessary to test this database model and try to filter time stamped device readings using SQL statements. Furthermore, we want to experiment with the first version of Time Abstraction API for SDA, described in the next section.

This schema has been tested using randomly generated data in the Java method **makeUpData**. For example, data for the tables **raw_datalogger** and **sda** have been generated ensuring that all the general assumptions about SDA data are true. Some of the assumptions that have been tested are:

- Overlapping cases
- Several shots for different owners happening simultaneously,
- Multi-set cases and single-set cases
- Non-concurrent cases, monopolizing SDA for a given owner, etc.

Time Abstraction for SDA

Class **SdaTimeInterval** from package **gov.fnal.sda.osda** represents the time abstraction for SDA. It has a logical level according to hierarchy:

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

That is, sets are owned by a case; cases are owned by a shot; shots are owned by an owner and an owner is held in a general slot. Logical path information is held in the **logicCoords** integer array:

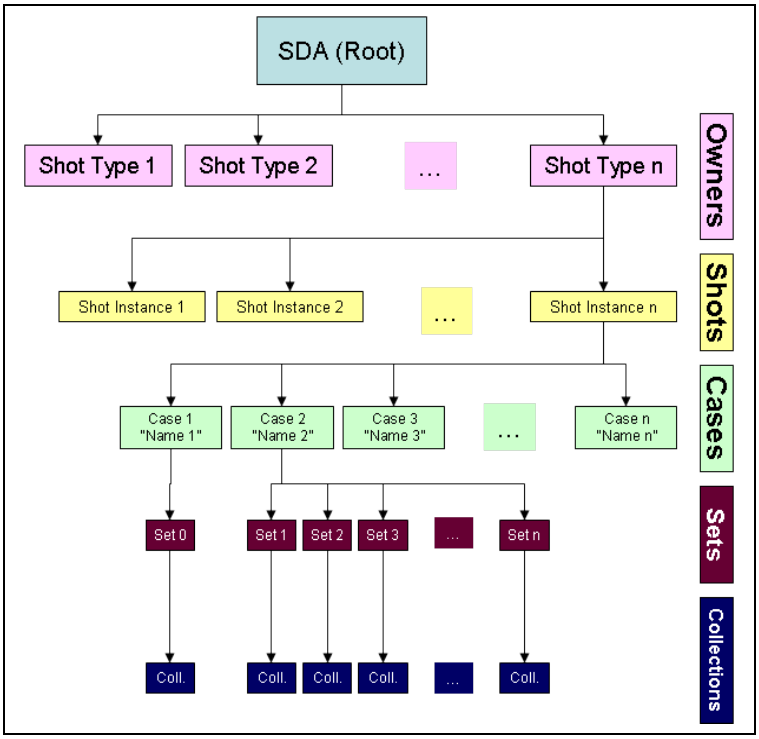
- logicCoords[0] → owner,
- logicCoords[1] → shot,
- logicCoords[2] → case,

- logicCoords[3] → set.

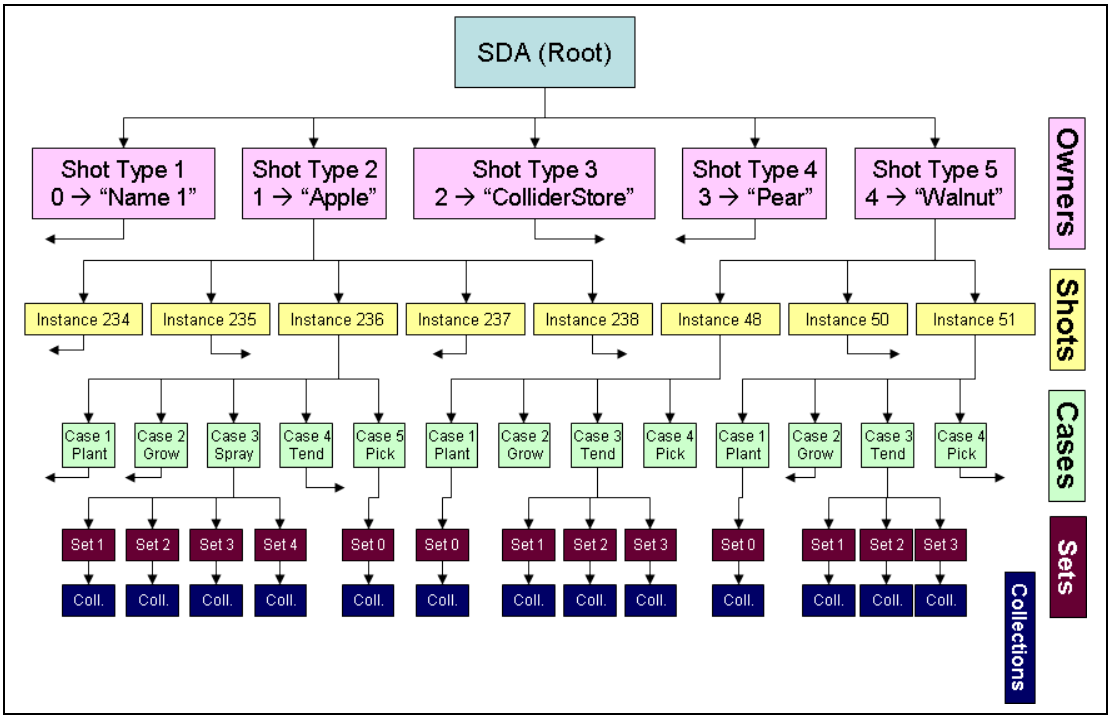
The value of -1 means that it has not been initialized. Therefore, this rule always holds:

if logicCoords[i] == -1 and i < j then logicCoords[j] == -1;

This figure shows this SDA tree layout schematically.



The next figure shows, as an example, how one instance of an SDA tree might look. Note that one can have as many "Owners" as necessary, but these are generally fixed. It is possible to add new "owners" as the need arises.



It is usually necessary to relate one shot instance to another. This connection is added onto the standard tree that is depicted here.

In addition to **logicCoords** there are **uniqueCoords** that may serve as database keys (recall that some shots and collections may be "stored over"). If the flag **valid** is not "stored over," **uniqueCoords** has similar meaning and rules as **logicCoords**. For convenience each **SdaTimeInterval** should have an **ownerName** and a **caseName**. The values of **logicCoords, uniqueCoords** and **names** in **SdaTimeInterval** must be *unmodifiable*.

Each **SdaTimeInterval** may have children **SdaTimeInterval**'s.

SdaTimeInterval defines these obvious getters:

- **getOwner**
- **getOwnerName**
- **getShot**
- **getShotName ...**

SdaTimeInterval define these boolean methods to determine the relative time placement of multiple instances of it:

- **includes,**
- **overlaps**
- **insideOf**

- **before**
- **after**
- **includes** (for long timestamp).

SdaTimeIterval also have several more complicated methods:

- **public SdaTimeInterval refine(int[]refinement)** - can be used for selecting all of the initial sets of the "high energy physics" cases for all shots within an 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 that contains only the shots, cases and sets for the timestamp, **ts**. It can be used for coloring 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 a child interval, if possible, in appropriate branch.
- **public boolean differentBranch(SdaTimeInterval interval)** determines that **this** and **interval** are on the same branch of the SDA tree .
- **public String toXml()** – serialize this object into XML.
- **public static SdaTimeInterval fromXml(String xml)** is not implemented yet, but it will be necessary when **toXml()** is utilized.

Results and Conclusions

The advantages of this approach are as follows:

1. Filling the **sda** table ("SDA timestamps and nothing else") requires minimal changes in just one implementation of the SDA interface class **gov.fnal.sda.db.xmldb.XmlDaqDataWriter**. The changes that are necessary are, roughly, six SQL queries. Thus, it comes for free with the normal SDA system.
2. **SdaTimeInterval** can be filled from an XML database as well. May be that the "offspring" time abstraction API will require another plugin for reading.
3. Filtering of device readings on the SQL level using a "SQL WHERE" condition and "JOINs" is possible and should have good performance. (We did not measure it because it is only in a test DB.)
4. The Hierarchical Time Abstraction API is relatively convenient and allows for all necessary timestamp / time intervals manipulations.
5. The Hierarchical Time Abstraction API allows an analog of the SDA Viewer to be written.
6. It is lightweight and does not include an XML database.
7. The amount of configuration needed to bring system up and running is small – only minimal attention from CERN people would be required.
8. Timestamps and intervals are available for other CERN programmers through an SQL query.

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

1. The system cannot guarantee that something will be read and stored for a particular collection. Our simulations show that holes like these occur regularly.
2. The ability to store big chunks of data that are collected at the right time ("sampled on the right event") is gone.
3. The system does not work as a backup for the normal datalogger.

The main advantage is #7. As for the disadvantages, we advocate for implementing normal SDA, with a relational DB or with DB XML, coupled by this **sda** table.

Appendix A. SQL queries to create a table.

The last version of this SQL file can be found in controls CVS in directory gov/fnal/sda/db/models as a_la_cern_sda.sql.

```
create table a_la_cern_raw_datalogger(
      devIndex     INT not null,
      ts           BIGINT not null,
      devReading   REAL not null
)
create unique index a_la_cern_datalogger_idx on a_la_cern_raw_datalogger( devIndex, ts )
create index a_la_cern_datalogger_ts on a_la_cern_raw_datalogger( ts )
create index a_la_cern_datalogger_devIndex on a_la_cern_raw_datalogger( devIndex )


create table a_la_cern_di2name(
      devIndex  INT not null,
      devName  VARCHAR(255) not null
)
create unique index a_la_cern_di2name_name on a_la_cern_di2name( devName )
create unique index a_la_cern_di2name_idx    on a_la_cern_di2name( devIndex )


create table a_la_cern_sda(
      owner     INT not null,
      shot      INT not null,
      theCase         INT not null,
      theSet          INT not null,
      file_idx    INT not null,
      coll_idx    INT not null,
      valid       BIT DEFAULT 1,
      tStart      BIGINT not null,
    tStop       BIGINT not null
)
create unique index a_la_cern_sda on a_la_cern_sda(  owner, file_idx, coll_idx )
create index a_la_cern_sda_h      on a_la_cern_sda(  owner, shot, theCase, theSet )
create index a_la_cern_sda_tstart on a_la_cern_sda(  tStart )
create index a_la_cern_sda_tstop  on a_la_cern_sda(  tStop )
```

```
create table a_la_cern_owner2name(
      owner     INT not null,
      ownerName   VARCHAR(255) not null
)
create unique index a_la_cern_o2n_o     on a_la_cern_owner2name( owner )
create unique index a_la_cern_o2n_n     on a_la_cern_owner2name( ownerName  )

create table a_la_cern_case2name(
      owner     INT not null,
    theCase     INT not null,
      caseName    VARCHAR(255) not null
)
create unique index a_la_cern_c2n_o  on a_la_cern_case2name( owner, theCase )
create        index a_la_cern_c2n_n  on a_la_cern_case2name( caseName  )

grant select,insert,update,delete on a_la_cern_raw_datalogger to public
grant select,insert,update,delete on a_la_cern_sda            to public
grant select,insert,update,delete on a_la_cern_di2name        to public
grant select,insert,update,delete on a_la_cern_case2name      to public
grant select,insert,update,delete on a_la_cern_owner2name     to public
```