Contents

Introduction

The Relational Blackboard (RBB) is an extension of the H2 Relational Database to support discrete events and timeseries data. In RBB, data are identified by sets of tags, which are name/value pairs. The RBB includes queries to find sets of co-occurring events, interpolate values for multiple timeseries at specified times, and find sequences of events matching specified patterns. The original motivation for RBB is as a knowledge base for cognitive systems and simulations. It is an active repository of information that can store larger-than-memory datasets and provide realtime notification when new data

become available. It supports rich, relational information (e.g. to reason about "classes" of objects) but generates fixed-length vectors suitable for machine learning algorithms. RBB is focused on data storage and retrieval, and not data analysis – its aim is to facilitate easy integration with external data analysis packages.

Features and Design Commitments

Languages

1. Applications may be written in a mix of languages
2. Applications may be written entirely without java, through SQL bindings. (A java runtime is still necessary to run the H2 database and RBB extensions).
3. Applications may be written entirely in java (without writing any SQL) by restriction to the RBB interface.
4. The RBB interface is not **H2**-specific; Java Client programs using the RBB do not need to use any H2-specific java interfaces or classes. However, the RBB interface *is* **SQL**-specific. For example, data retrievals returning a variable number of results return a java.sql.ResultSet. (The java.sql interface is already an abstraction over SQL implementations).
5. The RBB interface is implemented by H2RBB, which is H2-specific. This implementation of the RBB interface makes no effort to avoid H2-specific functionality because standard SQL does not specify how to implement functionality such as triggers and stored procedures that are necessary to implement the RBB interface.

Data

6. Data is stored exactly as it is received; resampling is performed on demand during retrieval.
7. Creating an RBB in a database does not preclude normal (non-RBB) use of the same database. RBB tables are given the prefix `RBB_`, and Timeseries observation tables are stored in the `RBB_TIMESERIES` schema.
8. One RBB corresponds to one H2 database. (Multiple RBBs in a single H2 database is not supported)

Runtime

9. Support for single-threaded applications, or multiple process applications on multiple hosts
10. Support for in-memory datasets, or for datasets too large for RAM.
11. Does not dictate the 'outer loop' (top level of control) in a program
12. The timeCoordinate functionality (for converting between different units of time) is optional; it does not complicate the API or significantly degrade execution speed in applications where all data is already in a single time coordinate system.

RBB Tutorial (tools.DrawTimeseries)

This section introduces RBB concepts using an included demonstration application, `tools.DrawTimeSeries`. This is not a tutorial for the RBB programming API; it simply illustrates RBB functionality. It was last tested with the cvs version tagged XXXX.

Configure Environment

The following environment variables used throughout this tutorial for convenience. They are not required by RBB.

```
cd <path>/RelationalBlackboard
export H2JAR=`pwd`/Libraries/h2-1.2.131/h2-1.2.131.jar
export RBBJAR=`pwd`/Projects/Core/Distribution/gov-sandia-cognition-rbb-core.jar
export RUNRBB="java -cp $H2JAR -jar $RBBJAR"
export MEMDB=jdbc:h2:tcp://localhost/mem:ExampleDB
```

Run DrawTimeseries

```
> $RUNRBB draw
```

This launches `tools.DrawTimeseries` (using an alias provided by `tools.Main`). A drawing window appears (Figure 1).

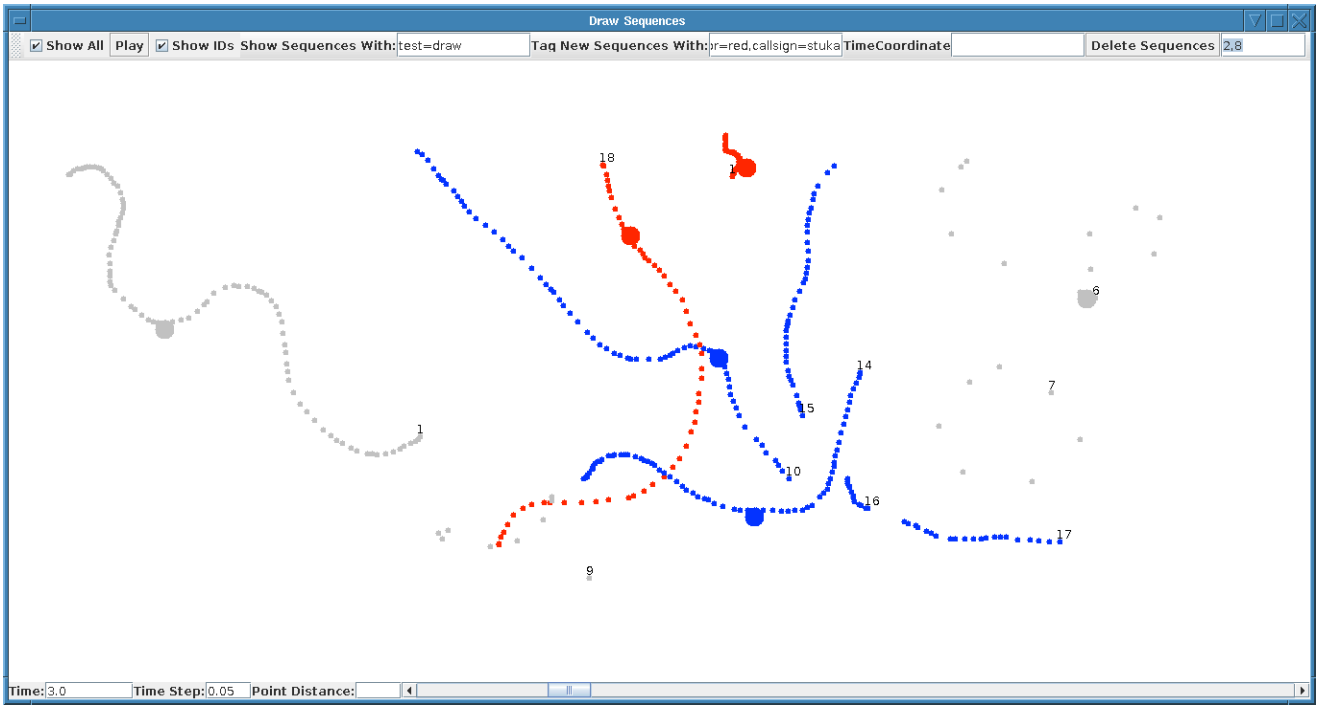

Figure 1: DrawTimeseries

Observations

`DrawTimeseries` is a simple drawing program. In RBB terms, clicking and dragging the left mouse button creates a **Timeseries** of **Observations** (x,y coordinates in this case). The observations in a timeseries are connected to depict the continuity of the timeseries. Each point drawn has a time as well as a position. To draw a timeseries, you must first click the "Play" button, which increments time in specified "TimeSteps." Pressing Play again stops time. You may then use the time slider bar (along the bottom edge) to move through time, replaying the timeseries forward or in reverse. By repeating the process you can create multiple timeseries, which may be concurrent (overlap in time), for example illustrating billiard balls colliding with each other. (However, `draw` does *not* simulate any physics such as momentum or collision detection, however). Toggling "Show All" causes observations to be displayed regardless of time, except that the current observation is shown with a larger dot.

RBB IDs

After drawing several timeseries, you might want to delete one or more of them. Each RBB timeseries has a unique ID number. Toggle "Show IDs" to display them. You can then delete a timeseries by entering the ID next to the "Delete Timeseries" and then clicking it. (You may also specify a range of IDs using a hyphen, or a comma-separated list of IDs and ranges, e.g. "3,4,5-12,7", although this is specific to `DrawTimeseries` and not a convention in RBB generally).

Tags

Using ID numbers to refer to timeseries is usually not very meaningful or convenient. IDs don't convey semantics (what the timeseries *mean* in a given application), nor provide a way to refer to groups of timeseries that are semantically related. RBB addresses these issues with Tags, which are simply <type>=<value> pairs. Each timeseries may have any number of tags, represented by a comma-separated taglist. Applications (such as `DrawTimeseries`) may assign meanings to tags, but only a few "magic" tags (documented in section Magic Tags) have any meaning to the RBB.

For example, assume we need to represent a battle in RBB, with both friendly and enemy airplanes and ground vehicles. `DrawTimeseries` recognizes the "color" tag, so we'll use "color=blue" for friendlies and "color=red" for hostiles. We'll make another tag type, "domain," to specify either "domain=ground" (for ground vehicles) or "domain=air" (for aircraft). In `DrawTimeseries`, taglists are specified in the textbox labeled "Tag New Timeseries With:". Each new timeseries is permanently associated with the tags in effect when the timeseries was created. Draw one or more timeseries with each of

the following tagsets, using the scrollbar to rewind the time to 0 before each one:

- `test=draw,domain=ground,color=blue,callsign=sherman`
- `test=draw,domain=ground,color=red,callsign=panzer`
- `test=draw,domain=air,color=blue,callsign=mustang`
- `test=draw,domain=air,color=blue,callsign=b29`
- `test=draw,domain=air,color=red,callsign=stuka`

Tagsets need not be unique, but timeseries should be given different tagsets if it will ever be necessary to distinguish between them. (Usually timeseries with identical tagsets represent intermittent observations of the same underlying object or physical process, such as locations recorded by a GPS receiver that occasionally loses reception).

Now you can use tagsets to restrict which timeseries are displayed by specifying a tagset in the textbox labeled "Show Timeseries With:". Of all the timeseries in the RBB, only those with *all* of the specified tags will be displayed. In RBB, tags used in this way are called **filter_tags**. Here are a few filter tagsets to try:

- `domain=air`
- `domain=air,color=blue`
- `callsign=stuka`
- `test=draw`

Each time a new tagset is specified, move the time scrollbar to update the display. With an empty filter tagset, no timeseries match and none will be displayed. In the example, all timeseries include "test=draw" so it is specified to display all timeseries.

The Delete Timeseries function also supports filter tags; try "domain=ground".

Interpolation

One of the basic issues addressed by RBB is that observations of concurrent timeseries may arrive asynchronously. It is often useful to create a "snapshot" by estimating the state of all timeseries at a given moment, and this is what `DrawTimeseries` actually displays.

Draw another timeseries after specifying a larger value for the "Time Step" input, such as 0.5 seconds. The observations will be spaced further apart. Now drag the time scrollbar slowly; the RBB interpolates positions between the observations, so the estimated position moves smoothly.

Now add `interpolate=prev` to the setting `Tag New Timeseries With`, and draw another timeseries. As you move the scrollbar, RBB will move from point to point, without interpolating positions. This is appropriate for discrete/categorical data.

Distributed Applications

RBB supports distributed applications. Multiple local and remote applications can read and write to the same RBB, and can subscribe for updates to events or timeseries that match a specified tagset. Several copies of `DrawTimeseries` may be run, and all will immediately reflect the actions performed in any of the others.

While leaving the previous copy of `DrawTimeseries` running, and using the previous values for H2JAR and RBBJAR, run one or more additional instances using this command:

```
> $RUNRBB draw $MEMDB
```

This is the same as before, but with an explicit database URL referring to the default in-memory database. All local instances of `DrawTimeseries` launched this way will show the same contents.

You can see how event-driven applications communicate through the RBB using telnet. While draw is running, try:

```
telnet localhost 1974
```

Then press <enter> once (this specifies an empty tagset for event filtering, so all events will be sent). Now draw a short timeseries in the draw program, and telnet will output data in the format:

```
eventCreated   13      418.9   1.7E308 test=draw,type=drawing
```

```
eventDataAdded 13      418.946 1.7976931348623157E308 test=draw,type=drawing RBB_TIMESERIES.S13    418.957
        965.0   177.0   -418.957

eventDataAdded 13      418.946 1.7976931348623157E308 test=draw,type=drawing RBB_TIMESERIES.S13    418.974
        966.0   179.0   -418.974

eventModified  13      418.946 419.02  test=draw,type=drawing
```

A notification is sent each time an event (or timeseries) is created, modified, or deleted.

Textual Data

RBB is based on a SQL database, but is designed to maintain the convenience of working with data in a simple, textual format. `tools.Get` retrieves timeseries matching a specified tagset from the running instance of `DrawTimeseries`, and prints them to the console:

```
> $RUNRBB get $MEMDB test=draw
```

The simple text format is convenient for data manipulation tasks such as plotting, creating a spreadsheet, or modifying and storing timeseries. The following command creates a function, 'translate,' to add 40 to the 'x' coordinate of timeseries. The next line retrieves data from the running instance of 'get', translates them, and stores the new timeseries, which appear immediately in `DrawTimeseries`:

```
> translate () { perl -e 'while(<>){@a=split /,/; if(@a==3){$a[1]+=40; print join(",",@a);} else {
print } }'; }
> $RUNRBB get $MEMDB test=draw | translate | $RUNRBB put $MEMDB
```

Web Client

RBB data is also available through the H2 webserver, which provides a console for SQL commands. This is a very convenient way to interactively develop SQL queries for an RBB. With of `DrawTimeseries` still running, open this URL in a web browser: http://localhost:8082

A login dialog appears. Enter the following:

```
JDBC URL:      jdbc:h2:tcp://localhost/mem:ExampleDB
User Name:     sa
Password:      x
```

Press 'connect,' then try the following queries:

```
select * from rbb_events;
call rbb_find_events('color=blue', null, null, null);
select * from rbb_timeseries.s1;
call rbb_timeseries_value(1, 0.725, null, null);
call rbb_delete_events('color=blue');
```

RBB Concepts

This section revisits the concepts from the tutorial, but is more detailed and definitional. The concepts build on each other and are presented from most basic to most advanced.

IDs

RBB_IDs are numeric identifiers for all RBB objects (tagsets, events, and strings). IDs are normally generated implicitly by functions such as RBB_CREATE_EVENT. They are generated from the SQL sequence RBB_ID by invoking `nextval('RBB_ID')`. IDs for all types of objects are generated from this sequence so the same ID cannot represent objects of different types. This helps detect bugs involving type errors, e.g. calling `RBB_ID_TO_STRING` on a Tagset ID instead of a String ID. The SQL datatype of a SQL timeseries is `IDENTITY`, and the java type is `java.lang.Long` (signed 64 bit).

RBB Strings

RBB makes frequent and repetitive use of variable-length strings. It is inefficient to search VARCHARs, so RBB assigns RBB_IDs to strings, and stores the string itself only the `RBB_STRINGS` table. The Java and command-line utility interfaces to

RBB mostly hide the use of IDs for strings, but using SQL directly exposes them. Conversion between strings and IDs is done with `RBB_ID_TO_STRING` and `RBB_STRING_TO_ID`.

Tags and Tagsets

Sets of Events and Timeseries are selected using **Tags.** A Tag may be applied to any number of Timeseries. A Tag has a Type and a Value. Tags are often used in combination to select Events: Type1=Value1,Type2=Value2... This specifies the set of Events with the specified value for all of the specified tags (regardless of additional tags the matching Events may have). The type and value are both strings.

Following are examples of tag sets that might be used in various applications:

EDRT Debrief:
Callsign=Falcon_11,Package=Falcon_1,Side=Friendly,Domain=Air,Platform=F16

Soccer:
Team=Lobos:Date=2010.05.29:Player=23:Half=1

RAKC Line Drawing Experiment (multiple experiments, institutions, subjects, and experiments):
University=ND:Subject=384:Experiment=MorphStimulus:Session=1:Condition=Jumble:Trial=3:Stroke=2

Magic Tags

Certain tag types and values have special meaning to the RBB software. These are known as "Magic Tags." These are:

- **end=<**1234.56>: Specifies the end time of an Event for `tools.Put`. This tag is removed and not stored with the Event.
- **interpolate=<interp_type>:** Specifies the appropriate type of interpolation for a timeseries when none is otherwise specified. Allowed values are:
  - **prev:** the previously observed value is assumed to remain constant until the next observation. Before the first observation, the value is assumed to be the value it was later observed to be.
  - **linear**: the value is interpolated linearly between the previous and next values. Before or after the first or last sample, the value is the first or last sample, respectively.
- **start_time=<**1234.56>: Specifies the start time of an Event or Timeseries for `tools.Put`. (For timeseries, this is useful to specify a start time before the first observation). This tag is removed and not stored with the Event/TimeSeries.
- **time=<**1234.56>: Specifies both the start and end time of an Event for `tools.Put`.

Creating Appropriate Tag Sets

Tags provide a flexible and powerful way to organize and search for events. However, with great power and flexibility comes great responsibility. In order to effectively use tags, one should be aware of how to create appropriate tagsets.

Many applications will be based around the concept of an "entity"; some object or individual that is acting through time and space. In this situation there will often be one timeseries (which describes the entities location in space and time) and several events that describe properties of the entity at different times.

As an example, consider a travelling photographer in New York City named Peter. Peter wanders about town taking pictures of events of significance. We can create a timeseries for Peters GPS coordinates through out the day. In addition, we can create several events that indicate *what* Peter is doing at each timestep (i.e, eating lunch, taking pictures of car crashes, fighting evil, etc). For instance, Peter can be at lat/long 40.748, 73.984, taking pictures of the Empire State Building. As we create these events, we need to decide on tags to use.

First, we need some tags that describe Peter himself. These are properties of Peter that are unique to him and won't change throughout the day:

FirstName=Peter,

LastName=Parker,

Occupation=Photographer,

Company=DailyBugle.

We can call these tags “entity ID tags” as they serve to uniquely identify an entity.

Now we need to create a tag the identifies the type of event that we are creating. We have at least two, one describes the position of Peter throughout the day, and the other describes what Peter is doing. We can create a tag called “variable” to capture this. For the event that describes Peters position we can use “variable=position”. For events that describe what peter is doing, we can use the value “variable=task”. We can call tags that describe the events as “event type tags”

The “variable=task” tag indicates that the associated event is used to describe what Peter is doing, but not the actual task. We can rectify this by adding tags describing specific things, such as “task=takepictures”, “phototopic=empirestatebuilding”, etc. We call these “event data tags”. Event data tags provide information about the event through the tag set rather than any associated data tables.

So let’s put all this together and create some events. First let’s create a position event.

Tagset	Start Time	End Time	Description
FirstName=Peter,LastName=Parker, Occupation=Photographer,Company=DailyBugle,variable=position	8:00am	5:00pm	This event is a “timeseries” that describes the lat/long of peter throughout the day.

Note that we did not use any “event data tags” because all the data for this event is stored in RBB as a separate table.

Now lets create some events that indicate what Peter is doing.

Event ID	Tagset	Start Time	End Time	Description
1	FirstName=Peter,LastName=Parker, Occupation=Photographer,Company=DailyBugle,variable=position	8:00am	5:00pm	This event is a “timeseries” that describes the lat/long of peter throughout the day.
2	FirstName=Peter,LastName=Parker, Occupation=Photographer,Company=DailyBugle,variable=task, Task=lunch	12:00pm	12:30pm	Lunch time!
3	FirstName=Peter,LastName=Parker, Occupation=Photographer,Company=DailyBugle,variable=task, Task=takepicture,picturetopic=empirestatebuilding	2:00pm	2:45pm	Taking pictures of the empire state building
4	FirstName=Peter,LastName=Parker, Occupation=Photographer,Company=DailyBugle,variable=task, Task=fightingevil,enemy=greengoblin	3:00pm	4:00pm	Fighting the Green Goblin

Note that Events 2-3 had all the same entity id tags. Thus, we can associate the different events. So if we search for the lat/long at time 3:00pm, we get data from event 1, and because we have the time, we can look at the event data tags of event 4 to figure out what peter was doing then.

Tag Representations

Each interface to RBB (Java, SQL, and command-line/textual) its own representation of Tagsets:

- The **Java interface** is "Tagset," which is implemented by imp.h2.H2Tagset.
- The **SQL** representation is a SQL array of strings, which alternates names and values: ('name1','value1', 'name2','value2'). The length of the array is always even. This representation is also used in the **impl.h2.statics** package, which implements the RBB stored procedures and is limited to receiving and returning SQL datatypes supported by H2, where a SQL array of strings is an Object[] containing String instances. (H2 stored procedures cannot receive other datatypes such as String[] or Object[][]).
- The command-line text representation is a string, "name1=value1,name2=value2…" A tag without an '=', e.g. "x", creates a tag with a null value, which matches all tagsets with that tag, regardless of value. Note: this representation is limited; it uses '=' and ',' as delimiters, and cannot represent tagsets in which any name or value contains a ',', nor any name contains a '='. The command-line utilities Get and Put also cannot process tagsets containing newlines.

Converting between Tagset representations

From \ To	Java	SQL	Text
Java		Tagset.toArray()	Tagset.toString()
SQL	Tagset.fromArray(Object[]) *or* RBB.newTagset(Object[])		H2STagset.toString(Object[])
Text	Tagset.fromString(String) *or* RBB.newTagset(String)	H2STagset.fromString(String)	

To retrieve a tagset as an Object[] from a SQL ResultSet, use H2STagset.fromResultSet(rs ,colName).
To store a tagset in a SQL PreparedStatement, use PreparedStatement.setObject(colIndex, Object[]). (h2 does not support PreparedStatement.setArray).

Events

In RBB, an Event is a Tagset associated with a start time, end time, and unique identifier. For example, the location of a bomb detonation might be specified as:

```
event=detonation,munition=mark82,time=1234.56,location=35.0341,-106.54485
```

The documentation in the section Textual Representation of Timeseries and Events has more detail on the textual representation of Events.

Data in the tag values have no special meaning to RBB. (For example, RBB doesn't support finding all detonation events within some bounding rectangle). When tags are not sufficient to store the data associated with an Event (e.g. binary data, structured data, or numerical data that must be queried efficiently) it is recommended to create a table for the extra data, joined by `RBB_EVENTS.ID`.

Event Listeners

To enable data-driven processing, a Java object can be registered for notification when any Event matching a specified Tagset is created, modified, or deleted, or when a data is appended to a table belonging to a matched Event (e.g. a Timeseries). This is done by implementing the `EventListener` interface and calling `RBB.addEventListener`. For more information, see `gov.sandia.cognition.rbb.impl.h2.H2EventTriggerTest`.

Timeseries

Observations (i.e. samples) of variables (e.g. positions) belong to Timeseries. (E.g in a soccer game, the successive positions of the ball form a timeseries, starting when play begins and ending when the ball goes out of bounds.) One might imagine using an Event to represent each observation, but (aside from being inefficient) this would not reflect the fact that subsequent observations of the same variable are related. The basic assumption of a timeseries is that values can be interpolated, because the observations are samples from an underlying continuous process.

In RBB, a Timeseries is a type of Event. Each Timeseries is a row in `RBB_EVENTS`, and a table of observations, `RBB_TIMESERIES.SN`. (`RBB_TIMESERIES` is a SQL schema and *N* is the ID of the Timeseries Event). The `START_TIME` and `END_TIME` values in `RBB_EVENTS` normally bracket the times of all Observations in the Timeseries. For example, an aircraft might take off at 10:00, transmit its GPS location every minute from 10:04 to 10:55, and land at 11:00.

See the RBB Programming Paradigms section for information on creating and manipulating Timeseries.

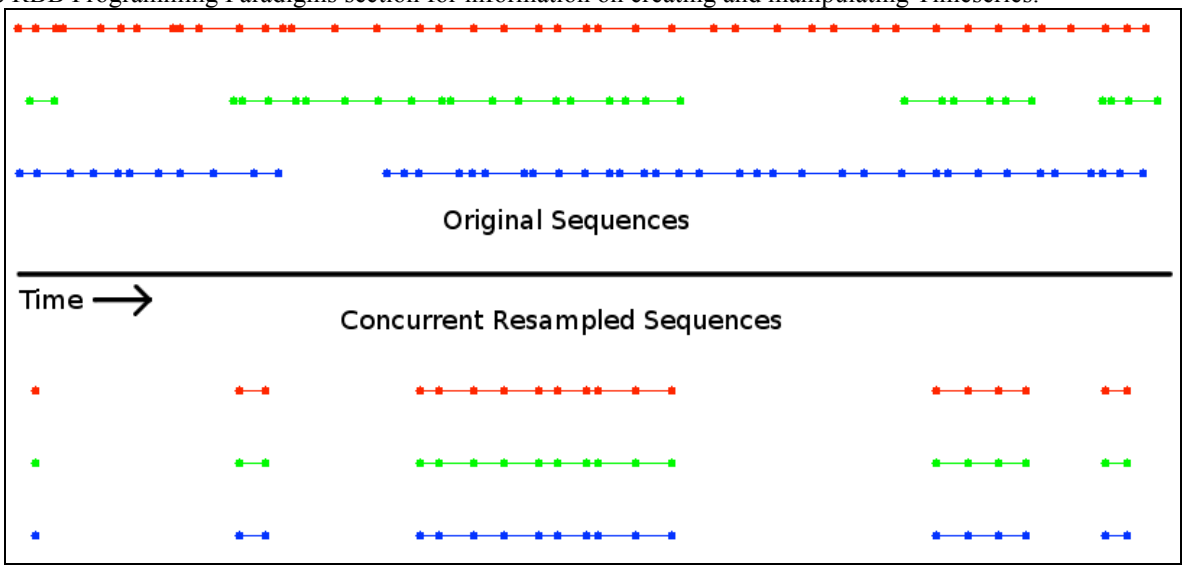

Figure 2: Concurrent Timeseries. Three sets of of timeseries (top) are resampled (bottom) during time intervals when they co-occur. The second and third sets of timeseries (green and blue) are resampled at the times of observation of the first set (red).

Time

RBB supports working with data stored in different time coordinates. For example, some data might be timestamped with UNIX time (seconds since Jan 1, 1970 UTC), while other data is timestamped with the number of milliseconds since a device was powered on, yet it may be necessary to retrieve a snapshot of the data reported simultaneously by both sources.

RBB_TIME_COORDINATES Table

Each RBB contains the table RBB_TIME_COORDINATES where the application defines time coordinates as linear transformations of a reference time, UTC. This table need not be populated if the application doesn't require any time conversions. It contains one row for each application-defined time coordinate. The row contains the linear transformation parameters (slope and offset) to convert from UTC into the specified time coordinate. In RBB, 'UTC' is simply a reference time coordinate for the application. It may refer to UNIX time, but could also designate the number of days A.D. – it is only important that all other time coordinates in that RBB are specified relative to the same reference time. Nonlinear transformations are not supported (e.g. months and years have varying lengths, so conversion of calendar dates is not possible).

The SLOPE parameter is the ratio between units of time in UTC and units of time in the coordinate. The INTERCEPT parameter is the offset from the time coordinate to UTC, in the units of the time coordinate.

Creating Time Coordinates

Time coordinates can be created by calling rbb_define_time_coordinate(tagset, slope, intercept), e.g.:
call rbb_define_time_coordinate('timeCoordinate=millisecondsUTC', 1000, 0);

Time coordinates may also be created by inserting a row into the RBB_TIME_COORDINATES table:
insert into rbb_time_coordinates values(rbb_string_to_id('timeCoordinate=millisecondsUTC'), 1000, 0, 0);

Here the first 0 is the INTERCEPT, and the value of the last parameter is not used. (The H2 database requires a value to be supplied for the TIME_COORDINATE_STRING_ID column, even though this column is computed automatically from the

TAGLIST_STRING_ID column, so the parameter is not used).

An application can define a time coordinate for UTC using an arbitrary name and the identity transform SLOPE=1, INTERCEPT=0:

```
call rbb_define_time_coordinate('timeCoordinate=secondsUTC', 1, 0);
```

Time Example

Consider an experiment in which each subject completes a single session (identified by the tag subject=n), which consists of several experimental conditions. UTC is defined as seconds since the UNIX Epoch. Eye tracking timeseries are timestamped in milliseconds since the Epoch (not UTC) and tagged with the name of a new time coordinate, millisecondsUTC: (example input to tools.Put)

```
subject=1,condition=1,variable=eyeTrackerXY,timeCoordinate=millisecondsUTC
1272820108000,400,325
1272820108010,391,325
1272820108020,368,251

subject=1,condition=2,variable=eyeTrackerXY,timeCoordinate=millisecondsUTC
1272820435000,400,325
1272820435010,391,325
1272820435020,368,251

subject=2,condition=1,variable=eyeTrackerXY,timeCoordinate=millisecondsUTC
1272823948000,400,325
1272823948010,391,325
1272823948020,368,251

subject=2,condition=2,variable=eyeTrackerXY,timeCoordinate=millisecondsUTC
1272824248000,400,325
1272824248010,391,325
1272824248020,368,251
```

millisecondsUTC is defined by a new row in RBB_TIME_COORDINATES (queries executed e.g. in the web client):

```
call rbb_define_time_coordinate('timeCoordinate=millisecondsUTC', 1000.0, 0);
```

This means there are 1000 milliseconds per second, and millisecondsUTC starts at the same time as UTC (intercept=0).

It may also be useful to document the UTC coordinate by defining it with the identity transformation parameters slope=1, intercept=0:

```
call rbb_define_time_coordinate('timeCoordinate=secondsUTC', 1, 0);
```

We might wish to retrieve data relative to the start of each session (e.g. to determine whether the subject's scan pattern changed between the 1st and 30th minute of performing the task). This is done by defining a new time coordinate for each session. The name of the timeCoordinate ('sessionSeconds') is the same for each subject, but this time coordinate is also conditioned on subject ID (since each subject starts at a different time), with the additional tag subject=n:

```
call rbb_define_time_coordinate('timeCoordinate=sessionSeconds,subject=1', 1, -1272820108);
```

The slope is 1 because the unit is seconds. The large number is the number of seconds before the session when UTC began. The intercept for session 2 is slightly larger in magnitude because it starts 64 minutes after session 1:

```
call rbb_define_time_coordinate('timeCoordinate=sessionSeconds,subject=2', 1, -1272823948);
```

Next we need to retrieve data relative to the start of each condition (e.g. to compare scan patterns between conditions with differing stimuli). This is done by defining a new time coordinate for each condition, which is conditioned on both 'subject' and 'condition':

```
call rbb_define_time_coordinate('timeCoordinate=conditionSeconds,subject=1,condition=1', 1, -
1272820108);
```

That is, the first condition of session 1 starts at the same time as the overall session. The second condition starts about 5 ½ minutes later:

```
call rbb_define_time_coordinate('timeCoordinate=conditionSeconds,subject=1,condition=2', 1, -
1272820435);
```

64 minutes after the start of session 1, the next subject begins session 2:

```
call rbb_define_time_coordinate('timeCoordinate=conditionSeconds,subject=2,condition=1', 1, -
```

```
1272823948);
call rbb_define_time_coordinate('timeCoordinate=conditionSeconds,subject=2,condition=2', 1, -
1272824248);
```

Furthermore, the experiment includes a parameter called 'difficulty' which changes on a fixed schedule relative to the start of each condition. The difficulty remains constant after being set (specified by the tag 'interpolate=prev'). The schedule is the same for all subjects. The difficulty for each condition is defined in a timeseries.

In condition 1 the task difficulty is increased each minute:

```
condition=1,variable=difficulty,timeCoordinate=conditionSeconds,interpolate=prev
0,20
60,50
120,100
999,100
```

In condition 2 the task difficulty is decreased each minute:

```
condition=2,variable=difficulty,timeCoordinate=conditionSeconds,interpolate=prev
0,100
60,40
120,10
999,10
```

As defined above, the time coordinate 'conditionSeconds' is conditioned on both 'subject' and 'condition', yet the 'difficulty' timeseries are tagged with 'condition' but not 'subject' (because the difficulty schedule is the same for all subjects). It is not possible to convert the timestamp from such a series to UTC without providing the additional context, 'subject'. That is, it is not logical to ask “what was the task difficulty at 10 AM last thursday” without reference to a particular subject; several subjects might have been performing different experimental conditions at that time.

Creating Time Coordinates for Events and Timeseries

In the example above, specifying a time coordinate for every session and condition is cumbersome. This process is automated by the function

```
rbb_define_time_coordinates_for_timeseries_combinations(String coordinateName,
        String filterTags, String joinTags, double timeScale)
```

It defines a new time coordinate for each combination of values of the 'joinTags' found in all events matching 'filterTags', e.g.:

```
call rbb_define_time_coordinates_for_event_combinations(
        'conditionMinutes', 'variable=eyeTrackerXY', 'subject,condition', 1.0/60);
```

This call creates a time coordinate named 'conditionMinutes' for each combination of the values of tags 'subject' and 'condition' – (1,1), (1,2), …, (2,1), (2,2)... The 0-time for each time coordinate (the INTERCEPT) is the start of the first event with each combination of tag values. The final parameter specifies the SLOPE – the ratio between UTC time units and units for the new time coordinate, e.g. 1/60 for UTC in seconds and a time coordinate in minutes.

A second example:

```
call rbb_define_time_coordinates_for_event_combinations('sessionSeconds', 'variable=eyeTrackerXY',
'subject', 1);
```

This call creates a time coordinate for each subject. For each value of 'subject', there are several eyeTrackerXY timeseries (one for each condition), but the start time of the earliest used.

Viewing the Contents of RBB_TIME_COORDINATES

The columns of RBB_TIME_COORDINATES are:

```
TAGLIST_STRING_ID  SLOPE  INTERCEPT  TIME_COORDINATE_STRING_ID
```

To show the table contents in human-readable form, execute the query:

```
select SLOPE, INTERCEPT, RBB_ID_TO_STRING(TAGLIST_STRING_ID) as TAGS from RBB_TIME_COORDINATES;
```

Following the running example from this section, we see a timeCoordinate for each combination of subject and condition has been created for the conditionSeconds.

SLOPE	INTERCEPT	TAGS

1000.0	0.0	timeCoordinate=millisecondsUTC
1.0	-1.272820108E9	subject=1,timeCoordinate=sessionSeconds
1.0	-1.272823948E9	subject=2,timeCoordinate=sessionSeconds
1.0	-1.272820108E9	condition=1,subject=1,timeCoordinate=conditionSeconds
1.0	-1.272820435E9	condition=2,subject=1,timeCoordinate=conditionSeconds
1.0	-1.272823948E9	condition=1,subject=2,timeCoordinate=conditionSeconds
1.0	-1.272824248E9	condition=2,subject=2,timeCoordinate=conditionSeconds
0.016666666666666666	-2.1213673916666668E7	condition=2,subject=1,timeCoordinate=conditionMinutes
0.016666666666666666	-2.1213668466666665E7	condition=1,subject=1,timeCoordinate=conditionMinutes
0.016666666666666666	-2.1213732466666665E7	condition=1,subject=2,timeCoordinate=conditionMinutes
0.016666666666666666	-2.1213737466666665E7	condition=2,subject=2,timeCoordinate=conditionMinutes

rbb_convert_time
The core time conversion function in RBB is
`rbb_convert_time(TIME, FROM_TAGS, TO_TAGS)`

Both FROM_TAGS and TO_TAGS must contain a tag 'timeCoordinate=abc' to specify the source or destination time coordinate. If either time coordinate is conditioned on other tags (e.g. 'subject=n' for sessionSeconds in the above example), these tags are also extracted from the taglist. If a tag is not present, the other taglist is consulted (TO_TAGS is the fallback for tags required for FROM_TAGS, and vice-versa).

Examples (using time coordinates created in the previous section):

```
TIME:      1234
FROM_TAGS: 'timeCoordinate=millisecondsUTC'
TO_TAGS:   'timeCoordinate=secondsUTC'
Result:    1.234
Comment:   1234 ms = 1.23 s

TIME:      0
FROM_TAGS: 'timeCoordinate=conditionSeconds,subject=1,condition=1'
TO_TAGS:   'timeCoordinate=sessionSeconds,subject=1'
Result:    0
Comment:   The session starts with the first condition

TIME:      0
FROM_TAGS: 'timeCoordinate=conditionSeconds,condition=2,subject=1'
TO_TAGS:   'timeCoordinate=sessionSeconds,subject=1'
Result:    327.0
Comment:   Condition 2 for subject 1 started 327s into the session

TIME:      0
FROM_TAGS: 'timeCoordinate=conditionSeconds,condition=2,subject=1'
TO_TAGS:   'timeCoordinate=sessionSeconds'
Result:    -327.0
Comment:   TO_TAGS has insufficient context for the 'sessionSeconds' time coordinate,
Comment:   (no 'subject') so FROM_TAGS is consulted and provides the default.

TIME:      0
FROM_TAGS: 'timeCoordinate=sessionSeconds'
TO_TAGS:   'timeCoordinate=conditionSeconds,condition=2,subject=1'
Result:    -327.0
Comment:   FROM_TAGS has insufficient context for the 'sessionSeconds' time coordinate,
Comment:   (no 'subject') so TO_TAGS is consulted and provides the default.

TIME:      0
FROM_TAGS: 'timeCoordinate=sessionSeconds,subject=1'
TO_TAGS:   'timeCoordinate=sessionSeconds,subject=2'
Result:    -3840.0
Comment:   Subject 1 started 3840s (64 minutes) before subject 2
```

Automatic Time Conversion
Normally rbb_convert_time is not invoked directly by applications; time conversion usually occurs implicitly to support queries over timeseries stored in varying time coordinates.

timeseriesValue functions (which estimate the value of a timeseries at a specified point in time) can perform time conversions through the timeCoordinate parameter, which supplies the FROM_TAGS argument for a call to RBB_CONVERT_TIME. The taglist of the specified timeseries supplies the TO_TAGS.

This example retrieves the difficulty of condition 1 (which happens to be RBB timeseries 1 in this example) as of a specified UTC time. This time conversion is subject-specific, so the subject is also specified. This is possible because FROM_TAGS can specify tag value defaults for the TO_TAGS.

```
call rbb_timeseries_value_prev(1, 1272820208, 'timeCoordinate=secondsUTC,subject=1');
```

The utility program RBB.util.GetTimeseries retrieves sets of co-occurring timeseries related by tags. Here is a simple invocation that retrieves all timeseries with the tag 'variable=eyeTrackerXY', just as they were entered. The UTC timestamps (e.g. 1.272820108E12) are hard to read:

```
> $JAVA RBB.util.GetTimeseries  $DB_SERVER variable=eyeTrackerXY


condition=1,subject=1,timeCoordinate=millisecondsUTC,variable=eyeTrackerXY

1.272820108E12,400.0,325.0

1.27282010801E12,391.0,325.0

...
```

An output timeCoordinate can be specified:

```
> $JAVA RBB.util.GetTimeseries -timeCoordinate 'timeCoordinate=sessionSeconds' $DB_SERVER
variable=eyeTrackerXY


condition=1,subject=1,timeCoordinate=millisecondsUTC,variable=eyeTrackerXY

0.0,400.0,325.0

0.009999990463256836,391.0,325.0

0.019999980926513672,368.0,251.0
```

However, the main purpose of this utility is retrieving several timeseries in parallel:

```
> $JAVA RBB.util.GetTimeseries -timeCoordinate 'timeCoordinate=sessionSeconds' $DB_SERVER
variable=eyeTrackerXY variable=difficulty,condition=


condition=1,subject=1,timeCoordinate=millisecondsUTC,variable=eyeTrackerXY
condition=1,interpolate=prev,timeCoordinate=conditionSeconds,variable=difficulty

0.000,400.0,325.0     0.000,20.0
0.010,391.0,325.0     0.001,20.0
0.020,368.0,251.0     0.020,20.0
…
condition=2,subject=1,timeCoordinate=millisecondsUTC,variable=eyeTrackerXY
condition=2,interpolate=prev,timeCoordinate=conditionSeconds,variable=difficulty
327.000,400.0,325.0   327.000,100.0
327.010,391.0,325.0   327.010,100.0
327.020,368.0,251.0   327.020,100.0
```

The output contains parallel timeseries (in columns) for the requested timeseries (variable=eyeTrackerXY and difficulty), with all times converted to sessionSeconds.

Deleting Time Coordinates

Time coordinates are deleted by

```
rbb_delete_time_coordinates(TAGLIST);
```

This deletes all time coordinates that match the tagset, e.g.:

```
call rbb_delete_time_coordinates('timeCoordinate=conditionSeconds,condition=2');
```

Deleting a time coordinate is allowed even if timeseries in the RBB are specified in that time coordinate, but time conversions for the timeseries will cause an error until a matching time coordinate is created.

Rules of Time

1. The Start and End Times in RBB_SEQUENCE_INFO are stored in the timeseries's native time coordinate - the same time coordinate as the data samples in the timeseries, which might not be convertible back to UTC without additional parameters.

2. findSequence:

   a. with a **null** timeCoordinate, no time conversion is performed. (This means the START and END arguments to findSequence are meaningless if the database contains timeseries stored in multiple timeCoordinates that match the filterTags).

   b. the timeCoordinate parameter is passed as a taglist that must include a timeCoordinate (e.g. "timeCoordinate=conditionSeconds") but may include other tags specifying time conversion parameters, which take precedence over the timeCoordinate parameters in the timeseries taglist.

   c. if given a timeCoordinate parameter, then the START and END parameters are interpreted as being in that time coordinate.

   d. conversion between time coordinates implies converting both to UTC, so sufficient context tags must be present for all timeCoordinates in timeseries matching the filterTags

   e. The START and END time in the timeseries infos returned by findTimeseries are in the requested output time coordinate.

   f. The tagset returned includes the timeCoordinate tag specified as a parameter, replacing the timeCoordinate tag attached to the timeseries.

3. Time conversions of maxDouble() and -maxDouble() don't change the value; they are treated as "infinity"

4. timeseriesValues:

   a. with a null timeCoordinate no conversions are performed

   b. the timeCoordinate parameter is passed as a taglist that must include e.g. "timeCoordinate=conditionSeconds" but may include other tags specifying time conversion parameters

   c. if given a timeCoordinate parameter, then START and END are interpreted as being in that time coordinate.

RBB Programming Paradigms

This section introduces alternate programming paradigms for RBB. More exhaustive API documentation is provided by Javadoc. It is intended that useful applications can be developed using any of these paradigms. Conversely, the paradigms may be mixed freely in an application; all operate on the same underlying representation, the H2 database.

This section describes each of the following tasks under each paradigm:

1. Creating an RBB
2. Creating a Timeseries
3. Adding an Observation to a Timeseries
4. Ending a Timeseries
5. Finding Timeseries
6. Retrieving data
7. Deleting timeseries
8. Event-driven processing

SQL

RBB is built on the H2 SQL database, so SQL is the most fundamental RBB interface. All the queries and commands in this section are in sql, so they may be executed from any language or environment with sql support. The other interfaces supplied by the RBB package (Java and command-line) build on the SQL interface. RBB extensions to the H2 database are

stored procedures in the package `rbb.impl.h2.statics`. The sql script "create_rbb.sql" in the source tree lists all the stored procedures available, and establishes the mapping from them to the Java static methods that implement them

Note the syntax for calling the H2 stored procedures in a query

```
call rbb_concurrent_events(('set=a','set=b', 'set=c'), 2.5, 100, null);
not
select rbb_concurrent_events(('set=a','set=b', 'set=c'), 2.5, 100, null);
```

Creating an RBB

Note: in most situations it is more convenient to create an RBB is through the command-line utility (see Command-Line Utilities section below), rather than through SQL directly.

The H2 database is created implicitly by connecting to a database that did not already exist. The H2 web console is a convenient way to do this. First, run the H2 server:

```
> cd <path>/RelationalBlackboard
> java -cp $RBBJAR:$H2JAR org.h2.tools.Server
```

The RBB jarfile `gov-sandia-cognition-rbb-core.jar` must be in the classpath; otherwise the stored procedures implementing RBB will not be available. Next create a database by connecting to the server. Example settings for the web console are:

```
JDBC URL:       jdbc:h2:tcp://localhost/mem:ExampleDB
User Name:      sa
Password:       x
```

The User Name 'sa' and Password 'x' are conventions in RBB. The JDBC URL specifies many attributes of the database; whether it is memory or disk-based, where it is stored on disk, etc. JDBC URL details are found in the online H2 documentation: http://www.h2database.com/html/features.html#database_url.

To create an RBB in the new database, execute the SQL command:

```
runscript from 'Projects/Core/Build/classes/gov/sandia/cognition/rbb/impl/h2/resources/create_rbb.sql';
```

This assumes the server was run from the RelationalBlackboard directory (the root of the source code directory hierarchy), as shown above; otherwise the sql script will not be found. The script creates a number of tables, e.g. RBB_DESCRIPTOR and RBB_STRINGS.

Creating a Sequence - rbb_start_timeseries(dimension, start_time, taglist)

'Dimension' is the number of data values in each observation, e.g. 2 for a timeseries of (x,y) coordinates.

```
set @id = rbb_start_timeseries(2, 1.0, 'tag1=value1,tag2=value2');
```

Adding an Observation to a Sequence - rbb_add_to_timeseries(SequenceID, time, (dim1, dim2,…))

```
call rbb_add_to_timeseries(@ID, 1.2, (10.1, 10.2));
```

Rows may also be inserted into a timeseries without using the rbb function. A final 'null' argument is required for the insert.[1]

```
insert into RBB_SEQUENCES.S1 values(1.3, 10.1, 10.2, null);
```

Ending a Sequence - rbb_end_timeseries(SequenceID, end_time)

```
call rbb_end_timeseries(@ID, 123.4);
```

By default, timeseries are open; they don't have an end time. (More precisely, the default end time is MAX_DOUBLE). In some cases this may be acceptable, but not if a subsequent timeseries (with a later start time) is intended to represent subsequent values of the same underlying dynamic variable.

[1] The extra column value is required for insert because the sequence table has an extra column whose value is computed automatically to be the negative of time. Indexing this column accelerates some common queries. Any value provided for this column is ignored, however not specifying a value causes an error. H2 does not allow specifying a default value for a computed column, either

Finding Timeseries

The RBB_SEQUENCE_INFO table can be queried directly to find timeseries IDs on the basis of start and end times:

```
select ID from rbb_timeseries_info where start_time > 0 and end_time < 20;
```

The RBB stored procedures support finding timeseries using tags in combination with time:

```
-- this is similar to select * from rbb_timeseries_info, but converts the taglists to readable strings:
call rbb_find_timeseries(null, null, null, null);
-- find only timeseries with at least the specified tags:
call rbb_find_timeseries('name1=value1,name2=value2', null, null, null);
-- find all timeseries that exist some time during the time interval [1,10]
call rbb_find_timeseries('name1=value1, 1, 10, null);
```

rbb_concurrent_events(filter_tags[], start_time, end_time, time_coordinate)

```
-- as used by the utility program that generated data for Figure 2
call rbb_concurrent_timeseries(('color=red','color=green','color=blue'), null, null, null);
```

Retrieving Observations

```
-- resample a timeseries at regular intervals
--  rbb_resample_timeseries_linear(SequenceID, start, timestep, end, time_coordinate)
call rbb_resample_timeseries_linear(40, 0, 1, 15, null);

-- get all observations within a time window
select * from rbb_timeseries.s40 where time >= 1 and time <= 15;

-- get the final observation at or before a specified time - not indexed; slow for big timeseries!
select * from rbb_timeseries.s72 where time  <= 500000 order by time desc limit 1;
-- same thing but indexed (fast) - this is what the minustime column is for
select * from rbb_timeseries.s72 where minustime  >= -500000 order by minustime limit 1;
-- same, but use rbb stored procedure
-- (can specify a time coordinate with optional last parameter)
call rbb_timeseries_value_prev(72, 500000, null);
-- same, except interpolate between neighboring values if no observation at exact time.
call rbb_timeseries_value_linear(72, 500000, null);
```

Deleting timeseries

It is not recommended to delete timeseries using SQL keywords directly (DROP TABLE… and DELETE FROM…) because deleting a timeseries affects several tables. Instead, use the stored procedures:

```
-- Delete a single timeseries by ID
call rbb_delete_timeseries_by_id(61);


-- Delete ALL timeseries matching the tagset (use with caution!)
call rbb_delete_timeseries('color=red');
```

Event-Driven Processing

There is no mechanism for event-driven (or "data push") applications in SQL, so programs receive notifications through a TCP socket. However a SQL call is used to start the TCP server in the RBB and retrieve its port number.

```
call RBB_START_EVENT_TCP_SERVER();
```

This returns the port number on which RBB is listening for TCP connections. This is commonly `1974`, but will vary if some other program was already using that port, or multiple RBBs are listening for connections simultaneously.

To determine the ip address of the server:

```
call RBB_GET_SERVER_ADDRESS();
```

This call is not necessary if the H2 server is running on the same host as the client program (simply connect using 127.0.0.1 or "localhost"). But if the client *may* run on a different host than the server, this allows the client to find the host without using any configuration information other than the SQL connection URL.

After establishing a tcp connection to this address and port, the client must send one line of text containing zero or more tagsets separated by semicolons and ending with a newline '\n' (not a \r\n pair), e.g:

```
test=x;test=y
```

The RBB will now send one line of text reporting any change to any event (or timeseries) matching any of the specified tagsets. The fields in the updates are separated by tabs. The updates are:

```
eventCreated   <id> <start> <end> <tagset>
eventDataAdded <id> <start> <end> <tagset> <schemaName.tableName> <newRowCol1> <newRowCol2>…
eventModified  <id> <start> <end> <tagset>
eventDeleted   <id> <start> <end> <tagset>
```

In each case, <start> and <end> are the start/end times of the event (or timeseries). As new samples are added to a timeseries, eventDataAdded will be emitted with the following extra columns:

```
RBB_TIMESERIES.S<id> <sampleTime> <sampleDim1> <sampleDim2>… <-sampleTime>
```

(The final column may be ignored – it is the time of the sample negated, which is an H2 implementation detail.)

RBB Stored Procedures[2]

For API documentation on the stored procedures, consult the javadoc of the corresponding java method, e.g. `RBB_DELETE_EVENTS` corresponds to `gov.sandia.cognition.rbb.impl.h2.statics.E2SEvent.delete`.

The mapping of parameters is normally straightforward, except the first parameter to the java function (a SQL database connection) is passed automatically by H2 and not specified when calling through SQL. The mapping from SQL to java data types is documented at http://www.h2database.com/html/datatypes.html

Note: all java classes are in package gov.sandia.cognition.rbb.impl.h2.statics	
Event (Class H2SEvent)	
addTags	RBB_ADD_TAGS_TO_EVENTS
create	RBB_CREATE_EVENT
defineTimeCoordinatesForEventCombinations	RBB_DEFINE_TIME_COORDINATES_FOR_EVENT_COMBINATIONS
delete	RBB_DELETE_EVENTS
deleteByID	RBB_DELETE_EVENT_BY_ID
deleteData	RBB_DELETE_EVENT_DATA
find	RBB_FIND_EVENTS
findConcurrent	RBB_CONCURRENT_EVENTS
findNext	RBB_FIND_NEXT_EVENT
findPrev	RBB_FIND_PREV_EVENT
findTagCombinations	RBB_EVENT_TAG_COMBINATIONS
getTagsByID	RBB_EVENT_TAGS
removeTags	RBB_REMOVE_TAGS_FROM_EVENTS
removeTagsByID	RBB_REMOVE_TAGS_FROM_EVENT_BY_ID
sequence	RBB_EVENT_SEQUENCE
setEnd	RBB_SET_EVENT_END
setEndByID	RBB_SET_EVENT_END_BY_ID
	RBB_START_EVENT_TCP_SERVER
Problem (Class H2SProblem)	
distance	RBB_DISTANCE
floats	RBB_FLOATS
RBB (Class H2SRBB)	
maxDouble	RBB_MAX_DOUBLE

[2] This table is generated by a script named `document_create_rbb` in the source tree. Comments in the script provide more information on updating the table.

negative	RBB_NEGATIVE
privateGetServerAddress	RBB_GET_SERVER_ADDRESS
privateSetLocal	RBB_PRIVATE_SET_LOCAL
schemaVersion	RBB_SCHEMA_VERSION
String (Class H2SString)	
find	RBB_STRING_FIND_ID
fromID	RBB_ID_TO_STRING
toID	RBB_STRING_TO_ID
Tagset (Class H2STagset)	
find	RBB_TAGSET_FIND_ID
findCombinations	RBB_TAGSET_COMBINATIONS
fromID	RBB_ID_TO_TAGSET
fromIDs	RBB_IDS_TO_TAGSETS
fromString	RBB_TAGSET_FROM_STRING
getAllTagTypes	RBB_GET_TAGSET_TYPES
hasTagsQuery	RBB_TAGSET_HAS_TAGS_QUERY
set	RBB_SET_TAGS
toID	RBB_TAGSET_TO_ID
toString	RBB_TAGSET_TO_STRING
Time (Class H2STime)	
convert	RBB_CONVERT_TIME
coordinateFromTagset	RBB_TIME_COORDINATE_FROM_TAGSET
defineCoordinate	RBB_DEFINE_TIME_COORDINATE
deleteCoordinates	RBB_DELETE_TIME_COORDINATES
fromUTC	RBB_TIME_FROM_UTC
toUTC	RBB_TIME_TO_UTC
Timeseries (Class H2STimeseries)	
addSample	RBB_ADD_TO_TIMESERIES
addSampleByID	RBB_ADD_TO_TIMESERIES_BY_ID
find	RBB_FIND_TIMESERIES
findNearest	RBB_FIND_NEAREST_TIMESERIES
isTimeSeries	RBB_IS_TIMESERIES
resampleLinear	RBB_RESAMPLE_TIMESERIES_LINEAR
start	RBB_START_TIMESERIES
value	RBB_TIMESERIES_VALUE
valueLinear	RBB_TIMESERIES_VALUE_LINEAR
valuePrev	RBB_TIMESERIES_VALUE_PREV
values	RBB_TIMESERIES_VALUES

Java

This section shows a simple java code listing that performs basic RBB functions. The Java bindings for RBB can be used instead of calling the RBB stored procedures through SQL queries directly. This simplifies coding somewhat (java objects are used instead of RBB ID's, reducing parameter count) and provides compile-time checking.

Although not shown here, the Java interface to RBB also provides the Tagset interface for constructing and manipulating tagsets (e.g. adding or removing name/value pairs and superset/subset testing).

```
import gov.sandia.cognition.rbb.*;
public class RBBTutorial
{
    public static void main(String[] args)
    {
        try
        {
            ////// Create an RBB
            // This creates the H2 database, and adds RBB functionality to it.
            RBB rbb = RBBFactory.getDefault().createRBB("jdbc:h2:mem:HelloWorldDB");

            ////// Create a Sequence
            Sequence seq = rbb.startSequence(1, 0, "test=tutorial,n=1");

            ////// Adding an Observation to a Sequence
            seq.add(2.0, 22.2f);
            seq.add(3.0, 33.3f);

            ////// Ending a Sequence
            seq.end(4.0);

            ////// Finding Timeseries
            Sequence[] seqs = rbb.findTimeseries("n=1", null, null, null);
            System.err.println("I found " + seqs.length + " timeseries!");

            ////// Retrieving Observations
            float[] x = seqs[0].valueLinear(3.5, null);
            System.err.println("My interpolated value at time 3.5 is " + x[0]);

            ////// Deleting timeseries
            int n = rbb.deleteTimeseries("test=tutorial");
            System.err.println("I deleted " + n + " timeseries");
        }
        catch (java.sql.SQLException e)
        {
            System.err.println("error: " + e.getLocalizedMessage());
        }

    }
}
```

Command-Line Utilities (Scripting)

This section documents executable programs (written in Java or otherwise) that share information through files or pipelines. The pipeline might manipulate the contents of the RBB, or output data into a plotting program or spreadsheet file.

Starting a Database Server

This section assumes environment variables are set per the "**Configure Environment**" section. An additional variable specifies a JDBC URL for a persistent database accessed through a server:

```
export SERVERDB=jdbc:h2:tcp://localhost//tmp/mydb
```

The server is launched with the command:

```
java -cp $RBBJAR:$H2JAR org.h2.tools.Server
```

Accessing the database through a server is not always necessary; a program can access a database in embedded mode (i.e. in-process) using a url such as `jdbc:h2:/tmp/mydb`, which typically executes faster if the program will execute many separate queries. However, in the command-line paradigm it is typical to run many short-lived 'utility' programs, and opening a large database takes *much* longer than connecting to a server that already has it open. Also, embedded database access is exclusive, precluding processing pipelines where multiple utilities access the blackboard concurrently (e.g. a pipeline that starts with RBB `get`, modifies the data to create new timeseries, and ends by re-inserting the new timeseries with RBB `put`).

Creating an RBB

The following command creates an RBB:

```
$RUNRBB create $SERVERDB
```

The underlying H2 database will be created if it did not already exist. If this command is run on an existing RBB, its RBB-specifc contents will be lost (non-RBB tables will not be affected).

Creating a Sequence, Adding an Observation to a Sequence, and Ending a Sequence

Sending a textual timeseries to RBB 'put' creates, populates, and ends the timeseries. The start and end times of the timeseries are assumed to be the times of the first and last observations in the timeseries. This input inserts two timeseries:

```
cat <<EOF | $RUNRBB put $SERVERDB

test=draw,color=pink,src=cmdline,id=1
1,150,100
4,450,400

test=draw,color=blue,src=cmdline,id=2
2,100,100
6,100,400

EOF
```

The following script creates one million observations of particles in Brownian motion (100 timeseries of length 10,000):

```
perl -e '@colors=qw(red green blue pink black orange gray); for(1..100) { print
"test=draw,color=",$colors[rand(@colors)],"\n"; $s=15; ($cx,$cy)=($x,$y)=(250,250); for my
$i(0..10000){ print join(",",$i/20,$x,$y),"\n"; $x+=rand($s)-$s/2; $y+=rand($s)-$s/2; } }' | $RUNRBB
put $SERVERDB
```

These can be replayed in DrawTimeseries (but do *not* select "Show All"):

```
$RUNRBB draw $SERVERDB
```

Finding Timeseries and Retrieving Observations

The 'get' utility retrieves timeseries, selecting them by tagsets. A basic example is retrieving timeseries by tag:

```
$RUNRBB get $SERVERDB test=draw
```

By specifying additional tagsets, concurrent observation timeseries are retrieved (see also Figure 2):

```
$RUNRBB get $SERVERDB color=pink color=blue
```

Output:

```
color=pink,src=cmdline,test=draw     color=blue,src=cmdline,test=draw

4.0,450.0,400.0                      4.0,100.0,250.0
```

In this case only one observation is generated, because the leftmost timeseries (id=1) was observed only once during the time both timeseries existed (from time 2 to time 4). The values for all but the leftmost timeseries are interpolated to the times of the samples of the leftmost timeseries.

All combinations of all matching timeseries are output, so the output of this script can be enormous if the specified tagsets match many timeseries.

Deleting timeseries

To delete all timeseries matching a taglist, use the 'delete' utility:

```
$RUNRBB delete $SERVERDB src=cmdline
```

Deletion is permenant, so it must be used with great caution. To view the taglists of all timeseries matching a taglist before deleting them, run the 'get' utility with the 'tagsonly' option:

```
$RUNRBB get -tagsonly $SERVERDB src=cmdline
```

Textual Representation of Timeseries and Events

– tools.Put and tools.Get

Textual Representation of a Timeseries
tag1=value1,tag2=value2,tag3=value3,start_time=123.456,end_time=678.897
t1,x1,y1
t2,x2,y2
t3,x3,y3

Parallel Timeseries:
Multiple timeseries can be represented in columns in the format:
tag=value,... tag=value,...
t1=x11,y11,.. t1=x21,y21,...
t2=x12,y12,.. t2=x22,y22,...
The columns are separated by one or more spaces or tabs. (Allowing multiple whitespaces between columns allows the columns to be presented neatly in parallel).

Using RBB With Matlab

This section shows how to import data from RBB to Matlab for numerical processing using the Database Toolbox for Matlab. This uses the SQL interface to RBB. (Matlab can call Java functions directly; todo – see whether that is more convenient than accessing through SQL).

This example assumes RBB Draw is running with the "-server" option so it created the in-memory ExampleDB database, and some timeseries have been drawn.

To connect to the RBB, in Matlab: (todo: should need to add rbb jarfile to javaaddpath also?)

```
javaaddpath('pathToH2JAR/h2-1.3.148.jar')

rbb=database('', 'sa', 'x', 'org.h2.Driver', 'jdbc:h2:tcp://127.0.0.1/mem:ExampleDB')
```

To find information about all the timeseries, call:

```
setdbprefs('datareturnformat','cellarray'); % this is the default, but might have been overwritten

fetch(rbb, 'select RBB_TAGSET_TO_STRING(TAGS), DATA_SCHEMA, DATA_TABLE from
rbb_find_timeseries(null,null,null,null)')
```

Note: One issue with using RBB through Matlab is that Matlab doesn't support the ARRAY SQL type, which RBB uses often. For example, RBB Tagsets are passed as arrays of strings. The above uses RBB_TAGSET_TO_STRING to convert the tagset array to a single string.

To find only the timeseries with the tags `test=draw`, call:

```
fetch(rbb, 'select RBB_TAGSET_TO_STRING(TAGS), DATA_SCHEMA, DATA_TABLE from
rbb_find_timeseries(RBB_TAGSET_FROM_STRING(''test=draw''),null,null,null)')
```

An simple example that retrieves the x,y coordinates of timeseries table S3:

```
setdbprefs('datareturnformat','numeric');

fetch(rbb, 'select C1, C2 from RBB_TIMESERIES.S3;')
```

To retrieve the timestamps as well (without two separate queries), use Matlab's database cursor object:

```
setdbprefs('datareturnformat','numeric');

c=exec(rbb, 'select TIME, C1, C2 from RBB_TIMESERIES.S3;')

c=fetch(c) % retrieve the data table from the ResultSet without opening a second cursor object

t=c.Data(:,1) % column 1 contains time

xy=c.Data(:,2:3) % columns 2 and 3 contain x, y coordinates.
```

This example resamples two concurrent timeseries (with IDs 3 and 6) to get snapshots at a fixed frequency of 0.1 hz.

```
setdbprefs('datareturnformat','numeric');

c=exec(rbb, 'select TIME, array_get(C1,1), array_get(C1,2), array_get(C2,1), array_get(C2,2) from
rbb_timeseries_values((3, 6), null, null, 0.1, null);')

c=fetch(c)

t=c.Data(:,1)

s3=c.Data(:,2:3)

s6=c.Data(:,4:5)
```

The query is a bit complex because it uses the sql function `array_get` to place each dimension of each timeseries in a separate column. If this is not done, each column of the ResultSet of `rbb_timeseries_values` contains a SQL Array, which is displayed in Matlab as a java object handle.

This example retrieves all the samples for timeseries with the tags "type=a" during the times that events with tags "type=b" also exist:

```
setdbprefs('datareturnformat','numeric');

% get the IDs of sets of events that co-occur

IDS=fetch(rbb, 'select ARRAY_GET(IDS,1), ARRAY_GET(IDS,2) from
rbb_concurrent_events((''type'',''a''),(''type'',''b'')), null, null, null, null)')

% retrieve timeseries values from the type=a timeseries events.

for i=1:size(IDS,1); A=fetch(rbb, sprintf('select ARRAY_GET(C1,1), ARRAY_GET(C1,2) from
rbb_timeseries_values((%d,%d),null,null,null,null);', IDS(i,1), IDS(i,2))); disp(A); end
```

Todo: figure out how to create an RBB Timeseries from a Matlab array.
Todo: establish how to use RBB batch on problem instances to do numerical processing with Matlab.

Object Orientation (Draft Section)

Each RBB Event is a fact that is true for a specified timespan (or in the case of a Timeseries, a measurement that is available for a given timespan). RBB primarily takes a relational approach, where sets of related facts are linked by intersecting sets of tags. There is no explicit representation of groups of facts – i.e., objects. In contrast, the memory state of a typical object-oriented program represents the state of a set of objects at the present moment. A common use of the RBB is to estimate the state of selected attributes of selected "objects" at a specified time in the past (or present). This section documents how to create and query "objects" in RBB.

OO Member Variable = RBB Event / Timseries

In the object-oriented paradigm a member variable assumes some value (for some period of time, which typically is not represented). In RBB, each new state of the member variable corresponds to an RBB Event. The state of the attribute is represented in one or more tags, while one or more other tags link the attribute to one or more parent objects.

RBB takes a bottom-up approach to constructing Objects – an Object exists whenever all the attributes for it have been set. The application performs a query to determine the set of Objects at a point in time, or is notified of the creation / destruction of Objects.

OO Class = RBB Problem Set

A Class is a pattern that specifies the attributes present in an Object instance. In RBB, a Problem Set specifies the relation among a set of Events that must hold for them to constitute a Problem Set. This specification can be queried to find the set of all objects, or installed as a trigger so the application is notified when a new RBB_Object exists.

OO Object = RBB Problem

On Object is a complete set of the attributes specified by a class, including their state at a point in time.

Scalability / Performance

This section gives guidance on the maximum practical sizes for RBB datasets and tips to improve performance.

Creating a Single-Tag Primary Key

Many functions in RBB must find a set of events from a tagset. This takes longer if the database contains a large number of distinct tagsets, or if the 'find' tagset contains several tags (each corresponds to a sql JOIN). If you are frequently finding the same tagsets, try combining all the values of that tagset into a single tag (or at least combining several tags often used in the same combination).

For example, H2SEvent.sequence(db, t, idTags, infoTags) calls 'find' on 'idTags' every time. Try replacing:

```
H2SEvent.sequence(db, 29,
'firstName=Bob,lastName=Jones,DOB=19601004',
'currentLocation=home');
```

with:

```
H2SEvent.sequence(db, 29,
'id=Bob:Jones:19601004',
'firstName=Bob,lastName=Jones,DOB=19601004,currentLocation=home');
```

The result is the same except each event has an extra 'id' tag.

Caching Events

RBBEventCache and RBBTimeseriesCache are useful when making repetitive queries whose filter tags are always a subset of some known tagset. They keep all events matching the tagset in memory in the client process. This cache is updated as events are created, updated, or deleted (even these updates are performed by other processes).

Timeseries (Table) Overhead

Each timeseries in RBB is a table. The H2 database (as of Oct 2010) uses substantial resources for each table in the database whenever the database is opened. As a result the practical limit on the number of timeseries in an RBB is on the order of 100,000. The remainder of this section is supporting data from a benchmark.

The following table shows runtimes and memory usage for RBBs with varying numbers of timeseries. The columns are:

NumTimeseries: a new RBB was created and populated with the specified number of very short timeseries (two, one-dimensional observations).
Heapsize: the minimum value of -Xmx <heapsize> used to populate and then open the RBB without raising an OutOfMemory exception. Tested values for -Xmx were: 100M 500M 1000M 2000M.
PutTime: CPU ("user") seconds to populate the RBB using RBB.put. This is the "user" time
OpenTime: CPU ("user") seconds to open and close the database with the H2 shell.
PutRate / OpenRate: NumTimeseries divided by **PutTime** and **OpenTime**, respectively.
DBSize: size of the database on disk, in megabytes.
SeqSize is the size of the same series of timeseries in textual (.seq) format.

The computer is a Core 2 Duo T9600 laptop @ 2.80GHz with 4GB RAM on AC power (not battery) with swap disabled.

NumTimeseries	Heapsize (MB)	PutTime (s)	OpenTime (s)	PutRate	OpenRate	DBSize (MB)	SeqSize (MB)
10	100	0.50	0.50	20.00	20.00	0.13	0.01
100	100	1.60	1.30	62.50	76.92	0.78	0.01
1,000	100	4.40	2.10	227.27	476.19	7.40	0.03
10,000	500	15.00	7.70	666.67	1,298.70	73.90	0.37
50,000	1,000	78.45	35.04	637.35	1,426.90	378.43	1.98

100,000	2,000	221.52*	78.86	451.43	1,268.10	759.22	3.98
150,000	2,000	740.57*	- **	202.70	-	1116	6.13
200,000	0	-	-	-	-	-	8.29

* For 100,000 and 150,000 timeseries the "real" runtime was about twice the "user" (cpu) time reported, perhaps indicating insufficient system memory.
**It is strange that a 150,000 timeseries database could be created with a 2000MB heap, but not subsequently opened; however that is what happened.

If an RBB Is Slow to Open

The underlying H2 database is slow to open if there are many tables in the database. One solution to this is to run a standalone h2 server (instead of opening the database embedded) and append ;DB_CLOSE_DELAY=-1 (including the semicolon) to the end of the connection URL. The standalone server will still be slow to open the database the first time, but after that clients can quickly connect to the server and access the RBB.

RBB Developer Information

This section is for developers of RBB itself (as opposed to developers of applications that use the RBB).

Source Code Package Structure

RBB contains the following packages:

`gov.sandia.cognition.rbb` – contains the (implementation-independent) java interfaces for RBB – RBB, Event, Tagset, etc.

`gov.sandia.cognition.rbb.impl` – placeholder for implementations of RBB – currently there is just one, H2, with no specific plans for more.

`gov.sandia.cognition.rbb.impl.h2` – H2 implementation of RBB. The classes directly in this package implement the RBB interface, typically by calling functions from `gov.sandia.cognition.rbb.impl.h2.statics`. It is preferable not to put any significant implementation code in this package, since it is not callable through SQL – instead, implement the functionality in the h2.statics package, and put a wrapper for it here. The class names in this package are prefixed with `H2`.

`gov.sandia.cognition.rbb.impl.h2.statics` – The actual implementation code. The class structure mirrors `gov.sandia.cognition.rbb.impl`. The code in this package consists of stored procedures for H2, so they are callable both in java directly (using the RBB java interface) and through SQL. Stored procedures in H2 are public static functions that accept and return datatypes recognized by H2. For example, functions in this package cannot return arrays of primitive types, and e.g. return Float[] instead of float[]. When called as stored procedures, H2 will return these values as Object[] whose actual element type is Float. The class names are prefixed with `H2S`.

`gov.sandia.cognition.rbb.impl.h2.resources` – Contains one file, `create_rbb.sql`. This is a sql script executed within an H2 database to add all the tables and stored procedures that constitute an RBB. This file is a useful reference because it is the definitive mapping between SQL procedures and the Java functions that implement them.

Use of PreparedStatements

PreparedStatements are used in RBB, primarily so special characters in strings will be properly escaped. PreparedStatemets are not saved between invocations, to avoid inadvertent sharing between threads and between multiple RBBs in the same process. A mechanism could be create to store PreparedStatements, per thread and per database connection.

Design Rationale

This section documents decisions made in the design of RBB

MINUSTIME

The computed "minustime" column in the rbb_data columns is present to support optimized queries for timeseries elements either before or after a specified time.

The following tests show results for various queries on a 1d timeseries with 1 million elements whose time and value step from 1 to 1,000,000. The query is to find the last update before time 666666.6 (i.e. 666666).

```
set @t = 666666.6;
Update count: 0
(1 ms)
```

– correct but not optimzed:

```
select * from rbb_data1 where seq_id=63 and time = (select max(time) from rbb_data1 where seq_id=63 and time < @t);
  TIME   MINUSTIME  SEQ_ID     X
666666.0 -666666.0        63        666666.0
(1 row, 1458 ms)
```

– gives correct result, and is optimized. This is what the rbb stored procedures do. This takes advantage of the fact that the query uses the index on minustime, which is stored in descending order of time. Strictly speaking this is an implementation detail of H2 and not guaranteed by SQL to give the correct result:

```
select * from rbb_data1 where seq_id=63 and minustime > -@t limit 1;
  TIME   MINUSTIME  SEQ_ID     X
666666.0 -666666.0        63        666666.0
(1 row, 7 ms)
```

– a refinement of the previous query, guaranteed to give the correct result. But H2 does not optimize it:

```
select * from rbb_data1 where seq_id=63 and minustime > -@t order by minustime limit 1;
  TIME   MINUSTIME  SEQ_ID     X
666666.0 -666666.0        63        666666.0
(1 row, 1777 ms)
```

– semantically equivalent to minustime > -@t above, but does not give the desired result because it uses the index on time which is in ascending order.

```
select * from rbb_data1 where seq_id=63 and -time > -@t limit 1;
  TIME   MINUSTIME  SEQ_ID     X
1.0       -1.0          63        1.0
(1 row, 0 ms)
```

TODO

This section is for developers to record desired improvements to RBB.

- Need a call to add a tag to a tagset, e.g. to a timeseries.