AUDIT TRAILS IN POLYHEDRA®
How certain features of the Polyhedra DBMS can be used to provide a selective audit trail mechanism for a Polyhedra database

Contents

Abstract

The Polyhedra® DBMS does not automatically produce an audit trail that would allow one to trace responsibility for each individual change to a database. However, it does provide sufficient mechanism to allow a database designer to produce such an audit trail for selected activities. This document contains a worked example that illustrates one way of configuring Polyhedra to do such auditing, and describes the facilities it uses to achieve this effect.

ENEA

Introduction

In application areas such as SCADA (**S**upervisory **C**ontrol **A**nd **D**ata **A**cquisition: industrial control systems for monitoring complex systems such as power plants, water/oil/gas/electricity distribution networks, train networks, refineries as well as more simple factory-floor applications), operators monitor the system state and can interact with the system to control its behaviour. In a water distribution network, for example, operators will receive alarm messages to warn them when critical conditions are being approached such as high water levels, and can cause sluice gates to be raised or lowered to alleviate the position. They can also configure the system - for example, to control when alarms are raised.

In such systems, it is often necessary to know which operator was responsible for actions such as acknowledging an alarm, adjusting an alarm point, or manipulating the position of a sluice gate or the control parameters for a pump. Indeed, in many industries this may be a regulatory requirement. The responsibility for producing the audit trail lies with the overall application, not with the individual components, though of course the component must not offer security loopholes.

Where there is a database at the heart of the SCADA system the operators would not have direct access to the DBMS; instead they will be clicking on on-screen buttons, for example, which will be sending preconfigured SQL commands to the database system. Consequently, the DBMS does not itself have to produce audit trails - but it simplifies the job of the application developers if they can take advantage of any security features the DBMS offers. In the case of Polyhedra, audit trails are not automatically produced - but there are three features that assist the database designer to perform selective generation of audit trails:

- User-based security and connection tracking
- Trigger code
- Time-series data capture

The remainder of this document illustrates how a database designer can use these features, both to implement a role-based security policy and also to ensure a long-term record is made of the underlying users responsible for particular changes to the database.

User-based security

Polyhedra provides a security mechanism based on the concept of tables being owned by users, and users being assigned access rights to tables and views - and also to individual columns in tables and views. Thus, if you have a table called, say, currency, defined by the following schema:[1]

```
create table currency
( persistent
, code      varchar (3) primary key
, country   varchar (20)
, name      varchar (20)
, usdollar  real
);
```

[1] In the schema definition, `persistent` is a table attribute saying that the table contents should still be available after the database is stopped and restarted; the alternative is `transient`, used for information that does not need to be durable. These keywords can also be used as column attributes - so if for example the usdollar attribute is being updated automatically by a feed from an exchange, it would probably be sensible to mark it as transient even though the rest of the table is persistent. As a further enhancement, it is also possible to mark `transient` attributes and tables as `local`, to stop the contents being replicated to a standby server or read-only replica.

… then you could allow everyone to read the table, but restrict updates to the usdollar column to a particular user, by the following commands:

```
grant select on currency to PUBLIC;
grant update on currency (usdollar) to exchangefeed;
```

By default, user-based security is disabled. To enable it, create a table called users, with a definition equivalent to the following:

```
create table users
( persistent
, name     large varchar primary key
, password large varchar
);
```

The next time the database server is started, the DBMS will automatically create extra tables to record the table and column privileges, and will create two special records in the users table for PUBLIC and SYSTEM, but with null passwords. All other users are automatically able to do anything that the PUBLIC user can do; the SYSTEM account has full access to the database, so your first action should probably be to set a password on that account!

```
-- password initially null, so can give any value when logging on.
login SYSTEM nopassword;

update users set password = 'y79BTg679vyi6kCFHG487v' where name = 'SYSTEM'; commit;
```

You can now create records for various users, and grant them appropriate privileges. Note that client applications have no privileges until they have logged on - and by default, users have no privileges apart from those they inherit from the PUBLIC account.

As well as the users table, you can also add a table called dataconnection:

```
create table dataConnection
( transient, local
, id          integer primary key
, machine     integer
, username    large varchar references users
, client_type varchar(12)
, env         varchar(16)
);
```

Once this table has been added,[2] a record is automatically created in it every time a connection is established; when a user authenticates him/herself, the username field is set appropriately, and cleared again when the user logs off. The client_type and env field are filled in with values supplied by the client application when it connects; it TCP/IP is used for the connection, then the machine field will hold the 32-bit IP number of the client. In fact, for flexibility we can derive a table from this, and tell the database to create records in the derived table when a user connects; this provides a clean way of adding extra columns to the table, and also enables us to add 'trigger code' as described in the next section. (You cannot attach triggers to special tables such as dataconnection.)

```
create table auditedDataConnection
( transient
, derived from dataconnection
);
```

(At this stage, we have no added any columns in the derived table, so it just has those it inherits from the base table; we shall add a column later, when we extend the example.)

[2] Or, more accurately, if this table is present when the database server is started up again.

Trigger Code: the CL module

The CL module in the DBMS is a powerful scripting engine that allows you to associate code with tables. You can specify the actions to be done when a record is inserted and when one is deleted, and also associate different 'methods' with individual columns. The CL programming language is rich in capabilities, with conditional flow of control and various forms of iteration statement. CL can read and write attributes of records, locate records, and create and delete records, and operates inline within the transaction which triggered it; it can even abort a transaction, and specify the error code and error message that should be sent to the client.

CL for role-based security

To illustrate what can be done with CL, we shall use it to replace user-based security by role-based security. First we shall add the following line to the definition of the users table, to hold information about the role of a user:

```
, role     large varchar references users
```

With this definition in place, we can then add some roles, users and privileges:

```
insert into users (name, password) values ('operator', 'PRIVATE');
insert into users (name, password) values ('manager',  'PRIVATE');
insert into users (name, password, role) values ('Tim',  'secret', 'operator');
insert into users (name, password, role) values ('Andy', 'passwd', 'manager');
insert into users (name, password, role) values ('Paul', 'N7ujyHn','SYSTEM');
grant select on currency to PUBLIC;
grant insert on currency to operator, manager;
grant delete on currency to operator, manager;
grant update on currency (usdollar) to PUBLIC;
```

Basically, roles are represented by entries in the users table with a null value in the role field; normal users will have an associated role, but no privileges. In this case, anyone can query currency and update the usdollar column; only the operator and manager roles are allowed to insert or delete records in this table (plus the system user, of course).

Now, we shall add a trigger to the 'username' column of the auditedDataConnection table, so that when a normal user - Tim, say - logs on he is automatically assigned the right role - in this case, the operator role. In essence, omitting various checks, all that is needed is the following:

```
script auditedDataConnection
    on  set username
        if exists username then
            if exists role of username then
                set username to role of username
        end if
    end set username
end script
```

(Of course, 'omitting checks' is not very sensible when we are dealing with security. We will add some sensible checks in a moment - but for the moment we shall concentrate on the bare minimum needed.)

Let us look at this code snippet in detail, for the benefit of those who have not previously seen our trigger language. The `script dataConnection` … `end script` grouping associates a number of functions and procedures ("methods") with the named table; the `on set username` … `end set username` section defines the method to be applied whenever the designated attribute is changed. When a user logs on or logs off, the security system will set the username field of the appropriate dataconnection record, and this will trigger the invocation of the corresponding method defined above. The method will run in the context of the modified object, so when it refers to a column name, the right cell of this record is read or written. In this example, the

code first checks to see if someone is logging on (by seeing if the username is not null) and if so it inspects the appropriate record in the 'users' table to see if a role is defined (by seeing if '`role of username`' is not null: the CL compiler recognises when a field is a foreign key reference, and treats it as a pointer). If username has been set to null, or no role is defined for the user who has logged on, nothing needs to be done; however, the remaining case (where username points at a 'users' record with a defined role) needs a bit more attention.

Assume Tim has logged on. What happens in this case is that the CL code sets the username attribute of the auditedDataConnection record to point at the 'operator' record in the users table. What happens now is that our method is kicked off again, recursively! This time through, the "exists role of username" test will return false, so the method will just drop through leaving the database unchanged. Control will now return to the outer invocation of the method, immediately after the 'set' statement. Once the method is complete, the transaction will terminate, and all subsequent operations by the client will be done with 'operator' privileges.

Of course, the above code is a bit too simple. To see why, consider what happens if a mistake has been made in populating the users table: suppose a record had been inserted as follows:

```
insert into users (name,password,role) values ('orouberos', 'pw', 'orouberos');
```

Will we get into an infinite loop if orouberos logs on? Well in this case, no, as the CL statement '`set username to role of username`' will set the username attribute to the same value as it had before… and as there is no change, the 'on set' method will not be triggered for a second time! However, now consider the following:

```
insert into users (name,password,role) values ('alpha', 'pw', 'omega');
insert into users (name,password,role) values ('omega', 'pw', 'alpha');
```

Logging in as either alpha or omega will trigger infinite recursion, terminated only by memory exhaustion or hitting the CL stack limit. As it happens, this is trivially prevented by the addition the following code snippet before the 'set' statement in the previous CL code:

```
                if exists role of role of username then
                    abort transaction "recursive roles!"
                end if
```

An alternative would simply to say that one does not swap to a role which itself has a role: this would leave the user logged on, but with no privileges.

A further enhancement to this example would be to record in the database the name of the user who actually logged on. First, we would add an extra field to the auditedDataConnection table…

```
, actual_user large varchar references users
```

Now, we can adapt our 'on set' method to set and clear this attribute at the appropriate times; at the same time we can ensure that no-one can log in as a password-protected role, even if they knew the password:

```
    on  set username
        if exists username then
            if exists role of username then
                set actual_user to username                    -- (1)
                if not exists role of role of username then
                    set username to role of username
                end if
            else if exists actual_user then                    -- (2)
                if role of actual_user <> username then
                    set username to role of actual_user
                end if
            else if exists password of username then           -- (3)
                forget username
            end if
        else
            if exists actual_user then forget actual_user      -- (4)
        end if
    end set username
```

Looking at the black, emboldened stuff in the above, the line labelled (1) sets actual_user to record the real user prior to the role change (which is suppressed if the role has a role). The ‘else if’ clause labelled (2) stops any role change other than to the role appropriate for the real user. The next ‘else if’ clause (3) detects if someone is logging on as a password-protected role whilst no real user is logged on: setting username to null allows the login to succeed, but the session will have no access rights. Finally, the code labelled (4) cuts in when username has been set to null, and it ensures that we forget who originally logged on.

Global functions and constants

As well as attaching methods to tables, CL also allows the database designer to define global functions and global constants. Global constants are not just evaluated at compile time: they are also re-evaluated at each subsequent database start-up, and therefore can depend on (and affect) the contents of the database at that time. Consider, for example, a database containing a table that had been set up as follows:

```
create table latest_event
( transient
, tag        varchar (20) persistent primary key
, timestamp  datetime
, username   varchar (20)
, event_type varchar (20)
, event_text varchar (80)
);
```

Suppose next that the CL attached to the database included the following:

```
constant reference latest_event currency_event = Find_latest_event ("currency")
constant reference latest_event user_event     = Find_latest_event ("user")

function latest_event Find_latest_event (string tag)
    local reference latest_event obj
    locate latest_event (tag=tag) into obj
    if not exists obj then create latest_event (tag=tag) into obj
    return obj
end Find_latest_event
```

At start up of the database service, the CL system will evaluate `Find_latest_event ("currency")` and `Find_latest_event ("user")` in order to assign a value to the global constants `currency_event` and `user_event`. The function uses the CL `locate` statement to find a particular record in the `latest_event` table; if it does not exist, the function creates the record. Other CL code can make use of `currency_event` and `user_event`, which will point at specific records in the database for the duration of the session.

Time-series data: the Historian module

In an embedded application where, say, the current values of sensors or shares are kept in-hand in a database, it is occasionally useful or important to know how the values are changing over time. In the UK, for example, water utility companies are required to keep certain historic information online for at least 90 days, and offline records for 10 years. In a system monitoring 10's of thousands of analogue sensors whose values are changing by the second, this can rapidly add up to a huge volume of data that can swamp a normal database, let alone an in-memory one!

Fortunately, there are characteristics to this kind of data - and in particular, to the way that it is used - that means we have been able to develop a special module in Polyhedra to handle this requirement. The historian submodule is part of the database server, and can be configured (by creating and updating records in certain tables) to set up a number of 'logging streams', each monitoring a given table with a chosen sample interval (measured in seconds), or 'on change'. For each logging stream, you can then specify, among other things:

- which columns are to be logged;
- the amount of disk space to be used as a circular buffer containing raw sample data, and likewise for time-based compressed data; and,
- the names of 'pseudo-tables' that can be queried to retrieve sample data.

It is thus possible, just by using standard SQL, to set up a series of configuration records to say:

- *"Monitor the table called analog, logging away every 5 seconds the values of the (floating point) attribute called 'value' and the (integer) attribute called 'status', but only for those records where the 'log_me' attribute is true.*
  - *Use a 500 megabyte file (organised as 100 blocks of 5 megabytes each) as a circular FIFO buffer of raw samples.*
  - *Use another 200 megabytes for time-compressed records at 1-minute granularity, and another 200 megabytes at one hour granularity; when producing time-compressed samples, record the min, max and average for the value column, and the bitwise AND and bitwise OR of the status field.*
  - *Raw samples can be retrieved through a table called fsample, time-compressed samples via a table called fsample_c."*
- *"Monitor the table called digital, logging away the (Boolean) value and (integer) status attributes on change, using a 200 megabyte file as a circular buffer of raw samples; allow samples to be retrieved through the dsample table."*

With such a set-up, one would retrieve information about earlier values by querying one of the special tables that had been set up:

```
select * from analog_sample
where name='J4RS' and timestamp > now()-hours(1)
```

```
select * from compressed_analog_sample
where name in 'J4RS,J4KT'
  and granularity='3600s'
  and timestamp>'05-JAN-2003 09:38:15'
  and timestamp<'19-JAN-2003 09:38:15'
```

The former retrieves the last hours-worth of samples for a given analog, the latter retrieves 2 weeks-worth of 1-hour time-compressed samples for two analog points. The historian is optimised for queries like these, and maintains its own indexing on the files to reduce the number of file accesses required.

The historian is not limited to logging numeric values; it can log fields of any type supported by Polyhedra. Thus, to record events, one notes that they are normally associated with objects already being represented in the

database; one simply adds an extra string attribute to store the most recent event associated with that object (and perhaps a datetime attribute to record a timestamp), and configure the historian accordingly:

- *"monitor the latestEvent and timeOfLatestEvent columns of the component table, using a 100 Mbyte file to hold the samples that are recorded on change, and allowing the samples to be retrieved through queries an a table called event."*

Offline archives

While the historian undoubtedly can extend the amount of data stored by a Polyhedra database, there are still limits on the amount of data storage - though this time it is because there are limits on the size of a file. On most 32-bit systems, there is a limit of 2- or 4-gigabyte on the size of a file that can be opened for random access; where data is changing rapidly and fine-grained samples are needed, 4GB is clearly not enough to hold many years of samples. To cope with this, the historian introduces the idea of archive and extract files.

At any time, you can instruct the historian to produce an archive file containing the oldest <n> blocks of un-archived data from the buffer file. A file of the indicated name is produced, and the blocks in the buffer file are marked as archived. An archive file can be brought back online by creating a configuration record in the database. If a query is made of a special historian table to retrieve samples, such as the example queries given earlier, then the historian does not limit itself to using the information in the main buffer file, but it also uses any applicable archive files that contain data for the relevant period. Archive files are in a machine-independent format, and are not tied to any particular database instance; consequently, archive files can be brought back online into any database with a similar historian configuration. This gives great flexibility in the way the historian is used: it is possible, for example, to use the historian on an embedded system as a data logger in a vehicle; when back at base, the archive files for recent journeys could be transferred (via a network, perhaps, or removable media such as flash cards or CD-R) to a workstation and then brought online for further inspection. For even further flexibility, the Polyhedra toolset provides a utility to convert archive files into XML, enabling easy export of the time-series data into a disk-based data system for detailed analysis.

Generating an audit trail

We now have introduced all the tools we need to implement an audit trail within Polyhedra. Suppose, for example, there is a currency table in the database (as defined earlier), and we want to be able to determine the person who creates and deletes records in this table. Assume also that we have set up role-based security, and have defined a table called latest_event as described in the section about CL global constants and global functions. We could now set up the historian to generate a sample whenever the 'timestamp' attribute of a latest_event record is altered:

```
insert into logcontrol
              (id,source,namecolumn,timestampcolumn, triggercolumn, enable)
        values          (2,   'latest_event', 'tag',       'timestamp',      'timestamp',
true);
insert into logcolumn (type, control, name,   sourcecolumn)
              values (0,    2,       'type', 'event_type');
insert into logcolumn (type, control, name,   sourcecolumn)
              values (0,    2,       'text', 'event_text');
insert into logdata (rate,fedfromrate,control,buffercount,buffersize,directory)
             values ('0s', NULL,      2,      50,        2000,      '.');
commit;
update logcontrol set raw='event' where id=2; commit;
grant select on event to manager;
```

… and attach a script to latest_event to define a procedure:

```
script latest_event
    on log_event (string type, string text)
        local reference dataconnection dc = GetConnection ()
        set event_type to type
        set event_text to text
        if not exists dc then
            set username to "-"
        else if exists actual_user of dc then
            set username to name of actual_user of dc
        else if exists username of dc then
            set username to name of username of dc
        else
            set username to "?"
        end if
        set timestamp to now ()
    end log_event
end script
```

With the above configuration, the historian will generate a sample whenever the timestamp attribute of a latest_event record is changed, and the log_event procedure will set the event_type, event_text, username and timestamp attributes of a latest_event record whenever it is called. The historian configuration also allows use to query the historical data by inspection the special table called 'event' - the contents of this table are not present in the in-store data, but are automatically generated from all the data that is available to the historian: this includes its in-memory buffers, the file it uses to hold samples in a round-robin fashion, and any archive files that have been brought online.

Now let us look at the following CL code:

```
script currency
    on  create
       log_event ( "creation", "currency record ('" & code & "') created." \
                 ) of currency_event
    end create

    on  delete
       log_event ( "creation", "currency record ('" & code & "') deleted." \
                 ) of currency_event
    end delete
end script
```

If this CL code is attached to the database, then whenever a record is created in the currency table, the log_event procedure is invoked on the latest_event record whose primary key is “currency”. This will set various fields on that latest_event record, including the timestamp - and altering the timestamp will in turn will trigger the historian to generate a sample. The record can be retrieved by querying the event table. We can similarly attach code to the data_connection table to log whenever connections are established and when people log on, and to the users table to record whenever users are created or deleted.

Conclusion

While Polyhedra does not by default perform any special security policy or automatically generate audit trails of all activities, it provides features that be used in conjunction to provide audit trails tracking selected activity:

- user-based security can be enabled to force people to log on if they are to access the protected data;
- a connection table and special CL functions allows activity to be associated with the client that initiated it;
- CL code can react to the events of interest to record who was responsible; and,
- the historian can make sure these records are available long-term, for online or offline analysis.

The fact that CL can be used to tie together disparate features of Polyhedra to enforce a security policy such as traceability indicates the power and flexibility of the overall Polyhedra solution.

Appendix: full example code

This appendix brings together and extends the code and data snippets given above into a complete example, with additional checks and with more types of events being recorded. In particular, the audit trail is extended to record when clients connect and disconnect, and when users log on and off. Clearly, in a production version, further actions would need to be checked and tracked (such as alterations to the users table) but it should be clear how the code could be adapted to cover such requirements.

The example is also available in machine-readable form to registered users of Polyhedra via the Polyhedra customer support desk, along with instructions on how to run it.

The schema

```
-- enable user-based security (from next restart onwards)

create table users
( persistent
, name     large varchar primary key
, password large varchar hidden
, role     large varchar references users     -- add a field for role of user
);
```

```
-- enable tracking of data connections (from next restart onwards)

create table dataConnection
( transient
, local
, id          integer primary key
, machine     integer
, username    large varchar references users
, client_type varchar(12)
, env         varchar(16)
);

create table auditedDataConnection
( transient
, derived from dataconnection
, actual_user large varchar references users  -- we record the 'real' user here
);
```

```
-- set up a table to hold the most recent event associated with a tag.

create table latest_event
( transient
, tag        varchar (20) persistent primary key
, timestamp  datetime
, username   varchar (20)
, event_type varchar (20)
, event_text varchar (80)
);
```

```
--  basic currency table

create table currency
( persistent
, code      varchar (3) primary key
, country   varchar (20)
, name      varchar (20)
, usdollar  REAL
);
```

Initial data

```
login SYSTEM nopassword;

-- set up some user accounts, including the default ones and some roles

update users set password = 'PRIVATE' where name = 'SYSTEM';
insert into users (name,password) values ('operator', 'PRIVATE');
insert into users (name,password) values ('manager',  'PRIVATE');

insert into users (name,password,role) values ('Tim',  'secret',  'operator');
insert into users (name,password,role) values ('Andy', 'password', 'manager');
insert into users (name,password,role) values ('Nigel','ND7ujm6yhn','SYSTEM');
commit;

-- allow everyone read access to the system catalogue
-- (note: all users inherit the 'public' rights).

grant select on tables         to PUBLIC;
grant select on attributes     to PUBLIC;
grant select on indexes        to PUBLIC;
grant select on indexattrs     to PUBLIC;
grant select on dataconnection to PUBLIC;

-- allow everyone to read and update the currency table, but only operators
-- and managers (plus, of course, SYSTEM) will be able to insert and delete
-- records.

grant select on currency to PUBLIC;
grant insert on currency to operator, manager;
grant delete on currency to operator, manager;
grant update on currency (usdollar) to PUBLIC;
```

```
-- log changes to the 'latest_event' table (all columns)

insert into logcontrol
       (id, source, namecolumn, timestampcolumn, triggercolumn, enable)
values (2,  'latest_event', 'tag',      'timestamp',     'timestamp',   true);
insert into logcolumn (type, control, name,   sourcecolumn)
               values (0,    2,       'type', 'event_type');
insert into logcolumn (type, control, name,   sourcecolumn)
               values (0,    2,       'text', 'event_text');

-- decide how much data to record.
insert into logdata (rate,fedfromrate,control,buffercount,buffersize,directory)
                        values ('0s', NULL,      2,      50,        2000,      '.');
commit;
update logcontrol set raw='event' where id=2;
commit;
grant select on event to manager;
```

```
--        Initial data for basic currency demo, including logging.

insert into currency values ('GBP','UK','Pound',0.67);
insert into currency values ('FRF','France','Franc',7.07);
insert into currency values ('CHF','Switzerland','Franc',1.67);
insert into currency values ('CAD','Canada','Dollar',1.49);
insert into currency values ('AUD','Australia','Dollar',1.72);
insert into currency values ('JPY','Japan','Yen',109.5);
commit;
```

CL code

The CL compiler allows the following sections to be defined in separate files, or all together in a single file. In a small example such as this, there is little point in using more than one file. For large projects, where the CL code can run to thousand of lines, it is often sensible to break the file up, with a separate file for the global constants and global functions that are used in many places, and then having a separate file for each group of related tables (holding the scripts for those tables plus the global functions and constants that are only used by those scripts).

Global functions and constants

```
function string NumberToDottedIPAddress (string s)
-- given a string holding a number,
-- convert it to the equivalent IP dotted address
    local integer n  = CharToNum (s)
    local string  res

    set res to bitAnd (255, bitShiftRight (n, 24)) & "." & \
               bitAnd (255, bitShiftRight (n, 16)) & "." & \
               bitAnd (255, bitShiftRight (n,  8)) & "." & \
               bitAnd (255, n)
    --debug "\tNumberToDottedIPAddress (" & s & ") yields" && res
    return res
end NumberToDottedIPAddress
```

```
function string httpDate (datetime d)
-- convert a datetime to the canonical
-- http format, for example...
--  Tue, 29 Apr 97 10:14:05 GMT
    return (item (1+The_weekday(d)) of "Sun,Mon,Tue,Wed,Thu,Fri,Sat") & \
        "," && get_DateTimef (d, "%02d %03M %04y %02h:%02i:%02s GMT")
end httpDate
```

```
-- scripts and constants associated with latest_event table

constant reference latest_event connection_event = Find_latest_event ("connection")
constant reference latest_event currency_event   = Find_latest_event ("currency")
constant reference latest_event user_event       = Find_latest_event ("user")

function latest_event Find_latest_event (string tag)

    -- whilst of course this function can be invoked by any method
    -- attached to this database, it is intended only for use in
    -- initialising the above constants; other methods can simply
    -- use the global constants whenever they need to access an
    -- attribute or method of the relevant objects

    local reference latest_event obj
    locate latest_event (tag=tag) into obj
    if not exists obj then create latest_event (tag=tag) into obj
    return obj
end Find_latest_event
```

Methods for the ‘latest_event’ table

```
script latest_event
```

```
    on  create
        log_event ("creation", "latest_event(" & tag & ") created")
    end create
```

```
    on  delete
        log_event ("deletion", "latest_event(" & tag & ") deleted")
    end delete
```

```
    on log_event (string type, string text)

        -- something interesting has happened; record the information here,
        -- together with timestamp and user info, so that the historian can
        -- generate an event record.

        local datetime                          dt = now ()
        local reference dataconnection          dc = GetConnection ()
        local reference auditedDataConnection adc

        set adc         to dc as auditedDataConnection
        set event_type to type
        set event_text to text

        -- who to record as the user? if the transaction is client-generated, the
        -- local variable adc will have been set non-null, and we can follow the
        -- pointer to get the relevant 'users' record, and from there get a user name.

        if not exists adc then
            set username to "-"
        else if exists actual_user of adc then
            set username to name of actual_user of adc
        else if exists username of adc then
            set username to name of username of adc
        else
            set username to "?"
        end if

        -- if we are doing lots of things needing auditing in the same
        -- transaction, the system clock may not have fine enough granularity to
        -- give them all distinct timestamps. This can cause a problem, as the
        -- historian is looking for changes in the timestamp column to decide
        -- when to generate a new sample. So, cheat a little:

        if timestamp < dt then
            set timestamp to dt
        else
            add microseconds(1) to timestamp
        end if
    end log_event
```

```
end script
```

Methods for handling connections.

```
script auditedDataConnection
```

```
    on  create
        log_event ( "connect" \
                  , "connection" && id & "established from" && \
                    NumberToDottedIPAddress (machine) \
                  ) of connection_event
        if exists username then check_user ()
    end create
```

```
    on  delete
        log_event ( "disconnect", "connection" && id && "closed." \
                  ) of connection_event
    end delete
```

```
    on  set username

        -- this attribute is set by the system when a user logs on,
        -- and cleared when he/she logs off. It will also be set by
        -- the check_user code below, when a logged-on user is
        -- swapped to the appropriate role.

        check_user ()
    end set username
```

```
    on check_user

        -- called when username is changed, or when the connection is
        -- first established. If username is defined, we need to see
        -- if we need to swap to the associated role. Apart from that,
        -- the main thing we are doing is generating appropriate log
        -- events.

        if exists username then
            if exists role of username then

                -- user with a role: report the logon, and change to the role

                set actual_user to username
                log_event ( "logon" \
                          , "connection" && id & ": user" && \
                            name of actual_user && "has logged on from" && \
                            NumberToDottedIPAddress (machine) \
                          ) of connection_event
                if not exists role of role of username then
                    set username to role of username
                    -- this will recursively invoke me!
                end if

            else if exists actual_user then

                -- we have a user without a role, but actual_user is set: \
                -- it might be a valid attempt to swap to the role for a user.
                -- Check if the role is right, otherwise force it to be right!
                if role of actual_user = username then
                    log_event ( "role change" \
                              , "connection" && id & ": user" && \
                                name of actual_user && \
                                "running with role" && name of username \
                              ) of connection_event
                else
                    -- someone is logging on as a role, while another user is logged on.
                    log_event ( "logon" \
                              , "connection" && id & \
                                ": someone tried to log on as" && \
                                name of username && "before " && \
                                name of actual_user && "logged off - ignored." \
                              ) of connection_event
                    -- set ourselves back to the right role for the real user
                    set username to role of actual_user
                end if
```

```
            else if exists password of username then

                -- someone has logged on as a password-protected role?
                -- log them off.
                log_event ( "logon" \
                          , "connection" && id & ": user/role" && \
                            name of username && "has logged on from" && \
                            NumberToDottedIPAddress (machine) \
                          ) of connection_event
                forget username

            else

                -- someone has logged on as a role that is not password protected
                log_event ( "logon" \
                          , "connection" && id & \
                            ": user/role" && name of username && \
                            "has logged on from" && \
                            NumberToDottedIPAddress (machine) \
                          ) of connection_event

            end if

        else if exists actual_user then

            -- a user, running with role privileges, has logged off
            log_event ("logoff" \
                      , "connection" && id & ": user" && name of actual_user && \
                        "has logged off." \
                      ) of connection_event
            forget actual_user

        else

            -- a user who logged on with the name of a role has logged off.
            log_event ("logoff" \
                      , "connection" && id & ": the user/role has logged off." \
                      ) of connection_event

        end if
    end check_user
```

```
end script
```

Methods for the currency table

```
script currency

    on  create
       log_event ( "creation" \
                 , "currency record ('" & code & "') created." \
                 ) of currency_event
    end create

    on  delete
       log_event ( "creation" \
                 , "currency record ('" & code & "') deleted." \
                 ) of currency_event
    end delete

    on  set name
       abort transaction "cannot alter name attribute of currency"
    end set name

end script
```

Sample Configuration File

When the main Polyhedra executables (such as rtrdb, the main executable that runs the server for a database, or sqlc, which provides a client interface allowing the user to enter SQL commands and see the results) are set off, they need to be configured to tell them how to operate. The server, for example, needs to know what ports to claim so that clients can connect to it, and it needs to be told where a saved database file can be found. Such information is provided by 'resource settings' that can be specified via the command line when setting off the executable, but it is much easier to gather them together in a configuration file – thus when the database server is fired of by a command line such as 'rtrdb db' it will look for a file called poly.cfg, and will then read the resource settings defined for the 'entry point' labelled db. The configuration file parser allows an inheritance mechanism to make it easy for different entry points to share resources, and as with CL there is a comment convention that allows the author to add explanations to the configuration file.

```
common:
data_service = 8001

-- entry points for use with rtrdb, the database server: empty, db and dbcl

empty:common
type = rtrdb
suppress_log         = false
suppress_dvi         = true

dbnocl:empty
load_file            = test.dat
dataconnection_class = auditedDataconnection

db:dbnocl
cl_library           = db.cl

-- an entry point for use with SQLC

sql:common
type                 = sqlc
echo_commands        = no
ft_enable            = yes
```