Event-driven fine-grained auditing with Informix

Jacques Roy

September 16, 2010
(First published October 21, 2004)

Learn how to use triggers to implement fine-grained auditing. This article covers the use of the datablade API to generate auditing events from transactions. It also introduces a feature called trigger introspection that allows you to create a generic auditing function that can be applied to any table. The article comes with example code that implements the solutions discussed.
Note: This article has been updated to include changes in the product since the article was originally published (14 Sep 2010).

IBM® Informix® allows you to generate events out of the execution of triggers through the use of callback functions. It is possible to write a generic function that can generate an audit trail of the information manipulated through SQL and captured through trigger.

This article describes how these facilities can be use in different context to generate auditing records.

Triggers

A *trigger* is an action that is executed when an operation is executed on a table. All versions of Informix have supported triggers on the following SQL operations: `INSERT`, `UPDATE`, and `DELETE`.

The Informix release version 9.2x added support for `SELECT` triggers, and Version 9.40 added support for "Instead of" triggers that allow you to replace the current operation with the one in the trigger. This effectively allows you to update a non-updatable view.

Trigger syntax

The syntax for triggers is described by the following simplified syntax diagrams.

In these syntax diagrams, you select one item from the list in braces ("{" and "}") separated by a pipe symbol ("|"). The expressions surrounded by brackets ("[" and "]") represent optional items that may be required depending on the previous choices. This will be clarified shortly with examples.

Listing 1. Sample trigger syntax

```
CREATE TRIGGER trigger_name
{INSERT | UPDATE | DELETE | SELECT} ON table_name
[REFERENCING OLD AS old_name] [REFERENCING NEW AS new_name]
[BEFORE [WHEN (condition)] (action)]
[FOR EACH ROW [WHEN(condition)] (action)]
[AFTER [WHEN (condition)] (action)][ENABLED | DISABLED]

CREATE TRIGGER trigger_name INSTEAD OF
{INSERT ON | UPDATE ON | DELETE ON} view_name
[REFERENCING OLD AS old_name] [REFERENCING NEW AS new_name]
FOR EACH ROW [WHEN(condition)] (action) [ENABLED | DISABLED]
```

You can choose from three types of actions. All of them could be used in a trigger. They are:

- **Before**: Execute this action once before the triggering action executes
- **For each row**: Execute after each row processed
- **After**: Execute once after the triggering action executed even if no rows were processed

Note that each triggering action includes an optional condition that evaluates if the action will be executed. This can be useful when you want to generate auditing records only for suspicious processing like salary changes of more than 10%, for example. Each type of action (before, for each row, after) can contain multiple conditions and multiple actions, including multiple actions per condition.

Listing 2 provides an example of a trigger creation from the Informix SQL syntax manual:

Listing 2. Sample trigger creation from the Informix SQL syntax manual

```
CREATE TRIGGER up_trigger
UPDATE OF unit_price ON stock REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN (post.unit_price > pre.unit_price * 2)
(INSERT INTO warn_tab VALUES (pre.stock_num, pre.order_num, pre.unit_price, post.unit_price,
CURRENT) );
```

This statement inserts a row in the `warn_tab` table if the new `unit_price` is more than twice as much as the old one.

The action does not have to be an SQL statement. It can also be either an `EXECUTE PROCEDURE` statement or an `EXECUTE FUNCTION` statement.

Processing rows

We saw in the syntax diagrams and in the example above that we can reference both the before and after images of the row being processed. It has to be noted that we cannot pass the row reference to a function or a procedure. For example, the following create statement fails:

Listing 3. Create statement that fails

```
CREATE TRIGGER tab1instrig INSERT ON tab1
REFERENCING NEW AS post
FOR EACH ROW (EXECUTE PROCEDURE do_auditing('INSERT', post))
```

You can work around this problem by defining a row as you pass the arguments to the function. Assuming a two-column table, the previous statement would become:

Listing 4. Successful create statement

```
CREATE TRIGGER tab1instrig INSERT ON tab1
REFERENCING NEW AS post
FOR EACH ROW (EXECUTE PROCEDURE do_auditing('INSERT',
         ROW(post.pkid, post.col2)::ROW(pkid integer, col2 varchar(30)) ) )
```

This example shows that it is not sufficient to create a row but we have to include its definition through the casting operator (`::`). Before we can address this problem, we have to first take a look at a few extensibility concepts introduced in Informix 9.x.

Object-relational features

Informix 9.x introduces object-relational features. A few of these features facilitate the implementation of auditing functions. They are the new data type `ROW` and the user-defined functions (UDFs).

The `ROW` data type can be equated to a table definition: It defines multiple columns that are grouped together into a tuple. `ROW` types can be either names or unnamed. For example, you can define a row type as follows:

Listing 5. Define row type

```
CREATE ROW TYPE zipcode_t (
  state  CHAR(2),
  code  CHAR(5)
);
```

We can create elements of that type using the `zipcode_t` name. We can also create unnamed row types as we did in the trigger example above. The unnamed row type was created with the expression:

```
ROW(post.pkid, post.col2)::ROW(pkid integer, col2 varchar(30))
```

A `ROW` type can be used to created type tables or as the data type for a column in a table. In the case of the `zipcode_t` type, it could be used in a table definition:

Listing 6. Table definition

```
CREATE TABLE customer (
  FirstName  varchar(30),
. . .
  zip   zipcode_t,
. . .
);
```

User-defined functions (or procedures) can accept `ROW` types as arguments. Here is an example of formatting the each row returned in XML format:

```
SELECT genxml2('customer', customer) FROM customer;
```

This statement passes a row as the second argument of the `genxml2()` function. This argument, being the same name as the table it selects from, represents a row from the customer table. What is passed as argument is an unnamed row type. For this reason, `genxml2()` defines a first argument that gives a name for the row. This is then used as the top-level name in the XML representation. For more information on generating XML from Informix, see the article "Generating XML from Informix" listed in the reference section at the end of this article.

A `ROW` type is self-describing. When a UDF receives a ROW as argument, it can find out the number of columns that are defined, their names, their types and their content. A UDF can be defined as receiving a generic row. At runtime, it can extract enough information from the row to decide on the type of processing.

Generating auditing records

With what we just learned about `ROW` types and UDFs, we can see that it is possible to create an auditing function that can be used for any tables in your database. If we are planning to write to an auditing table, we have to make sure that we match the audit table definition, no matter which table is being audited.

A simple approach is to generate the audit record in an XML representation. We could then use an audit table with the following format:

Listing 7. Audit table format

```
CREATE TABLE auditTable (
  id  SERIAL PRIMARY KEY,
  tabname VARCHAR(128),
  log  LVARCHAR(30000)
);
```

We can create a function, `do_auditing()`, that takes up to four arguments: the table name, the trigger type (`INSERT`, `UPDATE`, `DELETE`, `SELECT`), and the before and after image of the row.

Trigger introspection

Informix, Version 9.40.xC4 introduced a set of functions in the DataBlade API to retrieve context information from a UDF. The DataBlade API is the programming interface used to interface with the database server in "C." This implies that the trigger introspection facility can only be used if the function or procedure called in the trigger is written in "C."

From this point on, the example code will assume the use of the stores7 demo database. If we wanted to create an audit of inserts into the customer table, we could create a function `do_auditing1()` and use it in the `CREATE TRIGGER` as follows:

```
CREATE TRIGGER custinstrig INSERT ON customer
FOR EACH ROW (EXECUTE PROCEDURE do_auditing1() )
```

The `do_auditing1()` function retrieved the row information and any other information that could be useful for the auditing. The trigger introspection functions include:

- **`mi_integer mi_hdr_status()`**: the status returned indicates if the function is executing in a HDR environment and if it is executing on the primary or the secondary.
- **`mi_string *mi_trigger_tabname(mi_integer flags)`**: returns triggering table or view. The flag argument indicates the format of the table name: if it includes the schema name, owner name, and so on.
- **`mi_integer mi_trigger_event()`**: trigger information (operation, before/after/foreach/instead)
- **`mi_integer mi_trigger_level()`**: Nesting level of the trigger
- **`mi_string *mi_trigger_name()`**: returns the name of the trigger
- **`MI_ROW *mi_trigger_get_old_row()`**: before image of the row
- **`MI_ROW *mi_trigger_get_new_row()`**: after image of the row

With these functions, we can find out everything about the trigger or its context. We can now write the `do_auditing1()` "C" procedure to provide the database modifications auditing.

The first thing to do is make sure we are in a trigger and processing each row:

Listing 8. Making sure we are in a trigger and processing each row

```
trigger_operation = mi_trigger_event();
  if (trigger_operation & MI_TRIGGER_NOT_IN_EVENT) {
    /* not in a trigger! generate an exception */
    mi_db_error_raise(NULL, MI_EXCEPTION,
     "do_auditing1() can only be called within a trigger!", NULL);
    return;
  }
  /* Make sure this is in a FOR EACH type of trigger */
  if (0 == (trigger_operation & MI_TRIGGER_FOREACH_EVENT) ){
    /* not in a for each trigger! generate an exception */
    mi_db_error_raise(NULL, MI_EXCEPTION,
     "do_auditing1() must be in a FOR EACH trigger operation", NULL);
 return;
  }
```

Once we know we are in the right context, we can prepare a log record based on the type of operation executed. The following code excerpt illustrates how it can be done:

Listing 9. Preparing a log record

```
trigger_operation &= (MI_TRIGGER_INSERT_EVENT | MI_TRIGGER_UPDATE_EVENT | MI_TRIGGER_DELETE_EVENT |
  MI_TRIGGER_SELECT_EVENT);

  /* Call the appropriate function */
  switch (trigger_operation) {
    case MI_TRIGGER_INSERT_EVENT:
   pdata = doInsertCN();
   break;
. . .
```

Once the log record has been created, the last thing we have to do is insert it into the auditing table:

Listing 10. Insert into audit table

```
. . .
sprintf(psql, "INSERT INTO auditTable VALUES(0, '%s', '%s')",tabname, pdata);
sessionConnection = mi_get_session_connection();
ret = mi_exec(sessionConnection, psql, MI_QUERY_NORMAL);
. . .
```

For all the details of the `do_auditing1()` implementation, please consult the example code provided with this article.

Getting other useful information

The `do_auditing1()` function records the table name and the row being added, modified, or removed. The datablade API provides two functions to allow you to further identify the statements:

- `mi_get_database_info()`: Retrieve basic information such as database name and username.
- `mi_get_id()`: Retrieve either the statement id or the session id.
- `mi_get_transaction_id()`: Obtain the current transaction id.

You can also retrieve the username of the user executing the trigger by using the SQL built-in function `USER` (see the Informix 11.50 SQL Syntax manual, pages 4-71).

Transaction boundary

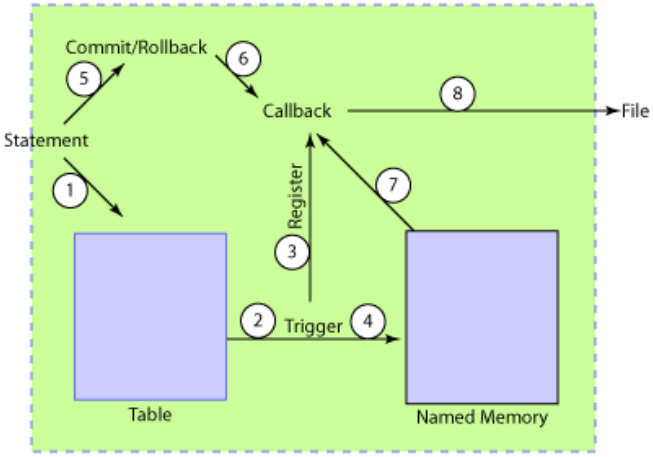
The implementation of `do_auditing1()` runs within the context of the current transaction. This means that if the transaction ends with a rollback, the record is removed from the auditTable table. This is fine for this implementation. In this case, if an auditing program needs to know about the changes to the auditTable table, it must go read the table at some time interval. Depending on how quickly it must react to these records, it could be every few seconds or, if it can be processed at a more leisurely pace, every few minutes or hours.

What if we want to write to a file outside the database or send the auditing record on a message queue? In this case, the operation cannot be completed until we know that the transaction has been committed. For this purpose, we need to be able to react to events.

Event processing

The DataBlade API provides ways to register callback functions that wait for specific events. This mechanism allows us to complete our auditing operation when the transaction completes. To implement the trigger, we follow the general approach illustrated in the following figure.

Figure 1. Event processing



When a statement executes (1), the trigger is called (2). The trigger registers a callback function (3) to write the result to a file. It also creates the auditing record and stores it in memory (4). This is a special type of memory available through the DataBlade API where you give it a name and can retrieve a reference to it by name.

A transaction can complete after one row is processed but can also finish after multiple rows. When this happens (5), Informix calls the callback function (6). The callback function can read the records that were saved in named memory (7) and write each record to a file (8).

The processing for this approach is similar to `do_auditing1()`. Let's call it `do_auditing2()`. It adds the creation of the named memory segment, the registration of a callback function, and the writing to files. The memory allocation is shown in Listing 11:

Listing 11. Memory allocation

```
sessionId = mi_get_id(sessionConnection, MI_SESSION_ID);
/* Retrieve or create session memory */
sprintf(buffer, "logger%d", sessionId);
if (MI_OK != mi_named_get(buffer, PER_SESSION, &pmem)) {
  /* wasn't there, allocate it */
  if (MI_OK != mi_named_zalloc(sizeof(NamedMemory_t), buffer, PER_SESSION, &pmem)) {
    mi_db_error_raise(NULL, MI_EXCEPTION, "Logger memory allocation error", NULL);
  }
}
```

We first retrieve the session ID to create a unique name for our named memory. We then try to get access to it. If it fails, this means that it is the first time this session called this trigger function. We then allocate the memory. Note that the third argument to the `mi_named_zalloc()` function is `PER_SESSION`. This means that the memory is allocated with a `PER_SESSION` duration. Once the user disconnects from the database server, the session disappears. Since the named memory was allocated on a `PER_SESSION` duration, the named memory is also freed.

The second addition to the code concerns the registration of a callback function.

Listing 12. Register the callback

```
/* Register the callback */
if (pmem->gothandle == 0) {
  cbhandle = mi_register_callback(NULL, MI_EVENT_END_XACT, cbfunc,(void *)pmem, NULL);
  if (cbhandle == NULL)
 mi_db_error_raise(NULL, MI_EXCEPTION, "Callback registration failed", NULL);
     pmem->gothandle = 1;
  }
}
```

This code registers a callback function called `cbfunc()`. A pointer to the named memory is passed as argument to `mi_register_callback()`. The `cbfunc()` function can the use it directly in its code. The function definition is:

```
MI_CALLBACK_STATUS MI_PROC_CALLBACK
  cbfunc(MI_EVENT_TYPE event_type, MI_CONNECTION *conn, void *event_data, void *user_data);
```

The key decision made in `cbfunc()` is to decide if we should write to the audit file. This is done with two tests. One test looks at the type of event (`MI_EVENT_END_XACT`) and the other to see if it ended in a commit (`MI_NORMAL_END`) or a rollback (`MI_ABORT_END`). Listing 13 provides some code illustrating this:

Listing 13. Testing if we should write to the audit file

```
if (event_type == MI_EVENT_END_XACT) {
. . .
change_type = mi_transition_type(event_data);
switch(change_type) {
  case MI_NORMAL_END:
   . . .
  case MI_ABORT_END:
  . . .
```

Note that even in the case of a rollback, the callback function must do some cleanup, removing all the records that were part of the transaction from the named memory.

The DataBlade API provides functions to write to operating system files. The callback function creates a unique file name to write the audit records stored in named memory.

Listing 14. Writing to operating system files

```
sprintf(buffer, "%s%d_%d.xml", LOGGERFILEPREFIX, pmem->sessionId, pcur->seq);
fd = mi_file_open(buffer, O_WRONLY | O_APPEND | O_CREAT, 0644);
ret = mi_file_write(fd, pcur->xml, strlen(pcur->xml));
mi_file_close(fd);
```

The fastpath interface

The DataBlade API provides functions, called the fastpath interface, to call another UDR. The description of this interface is beyond the scope of this article. You can find more information on the fastpath interface in the documentation provided in the reference section later in this article.

This interface could be use to call other functions such as the ones defined in the MQSeries® DataBlade.

What about Java?

The fine-grained auditing capability described in this article is better implemented in "C" so you can take advantage of the introspection feature that allows you to implement a generic function that works for any table. This does not mean that Java™ cannot be used for part of the processing.

The advantage of using Java user-defined functions or procedures is that you have access to all the capabilities of the Java environment. This includes communication classes such as socket connections, HTTP protocol, and so on.

To demonstrate one way to use Java in our auditing functions, consider a new function, `do_auditing3()`. This function provides the same processing as `do_auditing2()` but changes the callback function slightly.

Instead of using the DataBlade API functions to write to a file, this callback function used the fastpath interface to call a Java user-defined procedure that will write to a file. This Java function is defined as follows:

Listing 15. Java function

```
CREATE PROCEDURE writeFile(lvarchar, lvarchar)
EXTERNAL NAME
'audit_jar:RecordAudit.writeFile(java.lang.String, java.lang.String)'
LANGUAGE JAVA;
```

The first argument represents the file name and the second argument, the audit record. The callback function executes the Java procedure using the fastpath interface. It first finds a reference to the function and then executes it with the appropriate argument. This is demonstrated with the following code:

Listing 16. Finding and executing appropriate argument

```
fn = mi_routine_get(conn, 0, "writeFile(lvarchar, lvarchar)");
. . .
ret = mi_routine_exec(conn, fn, &ret, buffer, pcur->xml);
. . .
```

In the `mi_routine_exec()` function, the arguments `buffer` and `pcur->xml` are the arguments to the `writeFile()` function. The function reference fn must be release once we are done with it:

```
mi_routine_end(conn, fn);
```

Example code

This article comes with example code that implements all the functions described here. There are four functions that implement the different auditing schemes. They are:

- `do_auditing1()`
- `do_auditing2()`

- `do_auditing3()`
- `writeFile(lvarchar, lvarchar)`

The example code comes with two make files: `WinNT.mak` for the Windows® environment, and `Unix.mak` as a generic UNIX® makefile. The installation assumes that you have a directory named `$INFORMIXDIR/extend/auditing`. For more information, consult the `README.txt` file included with the example code.

Downloadable resources

Description	Name	Size
source code	auditing.zip	27KB

Related topics

- Informix 11.50 Information Center (IBM): Find information that you need to use the Informix family of products and features.
  - IBM Informix Guide to SQL: Syntax, Version 11.50 (SC23-7751-05): Understand the syntax of the statements, data types, expressions, operators, and built-in functions of the Informix dialect of the SQL language, as implemented in Version 11.50.
  - IBM Informix DataBlade API Programmer's Guide, Version 9.4 (SC23-9429-03): Understand how to use the DataBlade API functions.
  - IBM Informix DataBlade API Function Reference, Version 9.4 (SC23-9428-01): Understand the DataBlade API functions and the subset of Informix ESQL/C functions that the DataBlade API supports.
- "Generating XML from Informix" (developerWorks, August 2010): Learn about the extensibility features of IBM Informix to generate XML-formatted data. Sample code is included.