Firebird 3.0 DDL triggers

In Firebird 3.0 triggers can be written to execute when database objects are modified or deleted. A typical use is to block unauthorised users from performing these tasks.

The following is an excerpt from the Firebird 3.0.0 Alpha 1 Release Notes (23 July 2013) chapter, *Procedural SQL (PSQL)*:

DDL triggers

A. dos Santos Fernandes

Note: This feature was sponsored with donations gathered at the Fifth Brazilian Firebird Developers' Day

The purpose of a *DDL trigger* is to enable restrictions to be placed on users who attempt to create, alter or drop a DDL object.

Syntax pattern

```
<database-trigger> ::=
{CREATE | RECREATE | CREATE OR ALTER}
TRIGGER <name>
[ACTIVE | INACTIVE]
{BEFORE | AFTER} <ddl event>
[POSITION <n>]
AS
BEGIN
...
END

<ddl event> ::=
ANY DDL STATEMENT
| <ddl event item> [{OR <ddl event item>}...]

<ddl event item> ::=
  CREATE TABLE
| ALTER TABLE
| DROP TABLE
| CREATE PROCEDURE
| ALTER PROCEDURE
| DROP PROCEDURE
| CREATE FUNCTION
| ALTER FUNCTION| DROP FUNCTION
| CREATE TRIGGER
| ALTER TRIGGER
| DROP TRIGGER
```

```
| CREATE EXCEPTION
| ALTER EXCEPTION
| DROP EXCEPTION
| CREATE VIEW
| ALTER VIEW
| DROP VIEW
| CREATE DOMAIN
| ALTER DOMAIN
| DROP DOMAIN
| CREATE ROLE
| ALTER ROLE
| DROP ROLE
| CREATE SEQUENCE
| ALTER SEQUENCE
| DROP SEQUENCE
| CREATE USER
| ALTER USER
| DROP USER
| CREATE INDEX
| ALTER INDEX
| DROP INDEX
| CREATE COLLATION
| DROP COLLATION
| ALTER CHARACTER SET
| CREATE PACKAGE
| ALTER PACKAGE
| DROP PACKAGE
| CREATE PACKAGE BODY
| DROP PACKAGE BODY
```

Important rule

The event type BEFORE or AFTER#[BEFORE | AFTER] of a DDL trigger cannot be changed.

Semantics

1. BEFORE triggers are fired before changes to the system tables. AFTER triggers are fired after system table changes.
2. When a DDL statement fires a trigger that raises an exception (BEFORE or AFTER, intentionally or unintentionally) the statement will not be committed. That is, exceptions can be used to ensure that a DDL operation will fail if the conditions are not precisely as intended.
3. DDL trigger actions are executed only when committing the transaction in which the affected DDL command runs. Never overlook the fact that what is possible to do in an AFTER trigger is exactly what is possible to do after a DDL command without autocommit. You cannot, for example, create a table in the trigger and use it there.
4. With CREATE OR ALTER statements, a trigger is fired one time at the CREATE event or the ALTER event, according to the previous existence of the object. With RECREATE statements, a trigger is fired for the DROP event if the object exists, and for the CREATE event.
5. ALTER and DROP events are generally not fired when the object name does not exist. For the exception, see point 6.

6. The exception to rule 5 is that BEFORE ALTER/DROP USER triggers fire even when the user name does not exist. This is because, underneath, these commands perform DML on the security database and the verification is not done before the command on it is run. This is likely to be different with embedded users, so do not write code that depends on this.
7. If some exception is raised after the DDL command starts its execution and before AFTER triggers are fired, AFTER triggers will not be fired.
8. Packaged procedures and triggers do not fire individual {CREATE | ALTER | DROP} {PROCEDURE | FUNCTION} triggers.

back to top of page

Support in utilities

A DDL trigger is a type of database trigger, so the parameters -nodbtriggers (gbak and isql) and -T (nbackup) apply to them. Remember that only the database owner and SYSDBA can use these switches.

Permissions

Only the database owner and SYSDBA can create, alter or drop DDL triggers.

DDL_TRIGGER context namespace

The introduction of DDL triggers brings with it the new DDL_TRIGGER namespace for use with RDB$GET_CONTEXT. Its usage is valid only when a DDL trigger is running. Its use is valid in stored procedures and functions called by DDL triggers.

The DDL_TRIGGER context works like a stack. Before a DDL trigger is fired, the values relative to the executed command are pushed onto this stack. After the trigger finishes, the values are popped. So in the case of cascade DDL statements, when an user DDL command fires a DDL trigger and this trigger executes another DDL command with EXECUTE STATEMENT, the values of the DDL_TRIGGER namespace are the ones relative to the command that fired the last DDL trigger on the call stack.

Elements of DDL_TRIGGER context

- **EVENT_TYPE:** event type (CREATE, ALTER, DROP).
- **OBJECT_TYPE:** object type (TABLE, VIEW, etc).
- **DDL_EVENT:** event name (<ddl event item>), where <ddl_event_item> is EVENT_TYPE || ' ' || OBJECT_TYPE.
- **OBJECT_NAME:** metadata object name.
- **SQL_TEXT:** SQL statement text.

back to top of page

Examples using DDL triggers

Here is how you might use a DDL trigger to enforce a consistent naming scheme, in this case, stored procedure names should begin with the prefix SP_:

```
create exception e_invalid_sp_name 'Invalid SP name (should start with
SP_)';

set term !;

create trigger trig_ddl_sp before CREATE PROCEDURE
as
begin
    if (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME') not starting 'SP_')
then
        exception e_invalid_sp_name;
end!

-- Test

create procedure sp_test
as
begin
end!

create procedure test
as
begin
end!

-- The last command raises this exception and procedure TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
-- -Invalid SP name (should start with SP_)
-- -At trigger 'TRIG_DDL_SP' line: 4, col: 5

set term ;!
```

back to top of page

Implement custom DDL security, in this case restricting the running of DDL commands to certain users:

```
create exception e_access_denied 'Access denied';

set term !;

create trigger trig_ddl before any ddl statement
```

```
as
begin
    if (current_user <> 'SUPER_USER') then
        exception e_access_denied;
end!

-- Test

create procedure sp_test
as
begin
end!

-- The last command raises this exception and procedure SP_TEST is not
created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5

set term ;!
```

back to top of page

Use a trigger to log DDL actions and attempts:

```
create sequence ddl_seq;

create table ddl_log (
    id bigint not null primary key,
    moment timestamp not null ,
    user_name varchar(31) not null,
    event_type varchar(25) not null,
    object_type varchar(25) not null,
    ddl_event varchar(25) not null,
    object_name varchar(31) not null,
     sql_text blob sub_type text not null,
    ok char(1) not null
);

set term !;

create trigger trig_ddl_log_before before any ddl statement
as
    declare id type of column ddl_log.id;
begin
    -- We do the changes in an AUTONOMOUS TRANSACTION, so if an exception
happens
    -- and the command didn't run, the log will survive.
    in autonomous transaction do
```

```
    begin
        insert into ddl_log (id, moment, user_name, event_type, object_type,
                             ddl_event, object_name, sql_text, ok)
            values (next value for ddl_seq, current_timestamp, current_user,
                    rdb$get_context('DDL_TRIGGER', 'EVENT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'DDL_EVENT'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME'),
                    rdb$get_context('DDL_TRIGGER', 'SQL_TEXT'),
                    'N')
            returning id into id;
        rdb$set_context('USER_SESSION', 'trig_ddl_log_id', id);
     end
end!

-- Note: the above trigger will fire for this DDL command. It's good idea to
-- use -nodbtriggers when working with them!
create trigger trig_ddl_log_after after any ddl statement
as
begin
    -- Here we need an AUTONOMOUS TRANSACTION because the original
transaction
    -- will not see the record inserted on the BEFORE trigger autonomous
    -- transaction if user transaction is not READ COMMITTED.
    in autonomous transaction do
       update ddl_log set ok = 'Y'
       where id = rdb$get_context('USER_SESSION', 'trig_ddl_log_id');
end!

commit!

set term ;!

-- Delete the record about trig_ddl_log_after creation.
delete from ddl_log;
commit;

-- Test

-- This will be logged one time
-- (as T1 did not exist, RECREATE acts as CREATE) with OK = Y.
recreate table t1 (
    n1 integer,
    n2 integer
);

-- This will fail as T1 already exists, so OK will be N.
create table t1 (
    n1 integer,
    n2 integer
```

```
);

-- T2 does not exist. There will be no log.
drop table t2;
-- This will be logged twice
-- (as T1 exists, RECREATE acts as DROP and CREATE) with OK = Y.
recreate table t1 (
    n integer
);

commit;

select id, ddl_event, object_name, sql_text, ok
          from ddl_log order by id;
                     ID DDL_EVENT                 OBJECT_NAME
SQL_TEXT OK
====================== ========================== ======================
================ ======
                     2 CREATE TABLE               T1
80:3 Y
==============================================================================
SQL_TEXT:
recreate table t1 (
     n1 integer,
     n2 integer
)
==============================================================================
                     3 CREATE TABLE               T1
80:2 N
==============================================================================
SQL_TEXT:
create table t1 (
    n1 integer,
    n2 integer
)
==============================================================================
                     4 DROP TABLE                 T1
80:6 Y
==============================================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
==============================================================================
                     5 CREATE TABLE               T1
80:9 Y
==============================================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
```

==========