Data Definition Language (DDL)

In this chapter are the additions and improvements that have been added to the SQL data definition language subset in the Firebird 2 development cycle. Those marked as introduced in v.2.1 are available only to ODS 11.1 and higher databases.

Database triggers

Adriano dos Santos Fernandes

(**v.2.1**) A database trigger is a PSQL module that is executed when a connection or transaction event occurs. The events and the timings of their triggers are as follows:

CONNECT

- Database connection is established.
- A transaction is started.
- Triggers are fired; uncaught exceptions roll back the transaction, disconnect the attachment and are returned to the client.
- The transaction is committed.

DISCONNECT

- A transaction is started.
- Triggers are fired; uncaught exceptions roll back the transaction, disconnect the attachment and are swallowed.
- The transaction is committed.
- The attachment is disconnected.

TRANSACTION START

Triggers are fired in the newly-created user transaction; uncaught exceptions are returned to the client and the transaction is rolled back.

TRANSACTION COMMIT

Triggers are fired in the committing transaction; uncaught exceptions roll back the trigger's savepoint, the commit command is aborted and the exception is returned to the client.

Note: For two-phase transactions, the triggers are fired in the “prepare”, not in the commit.

TRANSACTION ROLLBACK

Triggers are fired during the rollback of the transaction. Changes done will be rolled back with the transaction. Exceptions are swallowed.

Syntax

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

<event> ::=
  CONNECT
    | DISCONNECT
    | TRANSACTION START
    | TRANSACTION COMMIT
    | TRANSACTION ROLLBACK
```

Rules and restrictions

1. Database triggers type cannot be changed.
2. Permission to create, recreate, create or alter, or drop database triggers is restricted to the database owner and SYSDBA.

Utilities support for database triggers

New parameters were added to gbak, nbackup and isql to suppress database triggers from running. They are available only to the database owner and SYSDBA:

```
gbak -nodbtriggers
isql -nodbtriggers
nbackup -T
```

See also:

Firebird and InterBase® command-line utilities

back to top of page

Global temporary tables

Vlad Khorsun

(**v.2.1**) Global temporary tables (GTTs) are tables that are stored in the system catalogue with permanent metadata, but with temporary data. Data from different connections (or transactions, depending on the scope) are isolated from each other, but the metadata of the GTT are shared among all connections and transactions.

There are two kinds of GTT:

- with data that persists for the lifetime of connection in which the specified GTT was referenced; and
- with data that persists only for the lifetime of the referencing transaction.

Syntax and rules for GTTs

```
CREATE GLOBAL TEMPORARY TABLE
  ...
  [ON COMMIT <DELETE | PRESERVE> ROWS]
```

Creates the metadata for the temporary table in the system catalogue.

The clause ON COMMIT sets the kind of temporary table:

ON COMMIT PRESERVE ROWS Data left in the given table after the end of the transaction remain in database until the connection ends. **ON COMMIT DELETE ROWS Data** in the given table are deleted from the database immediately after the end of the transaction. ON COMMIT DELETE ROWS is used by default if the optional clause ON COMMIT is not specified. **CREATE GLOBAL TEMPORARY TABLE** This is a regular DDL statement that is processed by the engine the same way as a CREATE TABLE statement is processed. Accordingly, it not possible to create or drop a GTT within a stored procedure or trigger.

Relation Type

GTT definitions are distinguished in the system catalogue from one another and from permanent tables by the value of RDB$RELATIONS.RDB$RELATION_TYPE:

- A GTT with the ON COMMIT PRESERVE ROWS option has RDB$RELATION_TYPE = 4
- A GTT with the ON COMMIT DELETE ROWS option has RDB$RELATION_TYPE = 5.

Note: For the full list of values, see RDB$TYPES.

Structural Feature Support

The same structural features that you can apply to regular tables (indexes, triggers, field-level and table level constraints) are also available to a GTT, with certain restrictions on how GTTs and regular tables can interrelate:

a. references between persistent and temporary tables are forbidden.

b. A GTT with ON COMMIT PRESERVE ROWS cannot have a reference on a GTT with ON COMMIT DELETE ROWS.

c. A domain constraint cannot have a reference to any GTT.

Implementation notes

An instance of a GTT - a set of data rows created by and visible within the given connection or transaction - is created when the GTT is referenced for the first time, usually at statement prepare time. Each instance has its own private set of pages on which data and indexes are stored. The data rows and indexes have the same physical storage layout as permanent tables.

When the connection or transaction ends, all pages of a GTT instance are released immediately. It is similar to what happens when a DROP TABLE is performed, except that the metadata definition is retained, of course. This is much quicker than the traditional row-by-row delete + garbage collection of deleted record versions.

Note: This method of deletion does not cause DELETE triggers to fire, so do not be tempted to define Before or After Delete triggers on the false assumption that you can incorporate some kind of “last rites” that will be executed just as your temporary data breathes its last!

The data and index pages of all GTT instances are placed in separate temporary files. Each connection has its own temporary file created the first time the connection references some GTT.

Note: These temporary files are always opened with Forced Writes = OFF, regardless of the database setting for Forced Writes.

No limit is placed on the number of GTT instances that can coexist. If you have N transactions active simultaneously and each transaction has referenced some GTT then you will have N instances of the GTT.

See also:

Table

back to top of page

Views enhancements

D. Yemanov

A couple of enhancements were made to view definitions in v.2.1:

Use column aliases in CREATE VIEW

Feature request CORE-831

(**v.2.1**) Column aliases can now be processed as column names in the view definition.

Example

```
CREATE VIEW V_TEST AS
  SELECT ID,
         COL1 AS CODE,
         COL2 AS NAME
  FROM TAB;
```

See also:

View

back to top of page

SQL2003 compliance for CREATE TRIGGER

A. dos Santos Fernandes

Feature request CORE-711

(**v.2.1**) Alternative syntax is now available for CREATE TRIGGER that complies with SQL2003.

Syntax patterns

Existing form:

```
create trigger t1
  FOR atable
  [active] before insert or update
as
  begin
     ...
  end
```

SQL2003 form:

```
create trigger t2
  [active] before insert or update
  ON atable
as
  begin
    ...
  end
```

Note the different positions of the clause identifying the table and the different keywords pointing to the table identifier (existing: FOR; SQL2003: ON).

Both syntaxes are valid and are available also for all CREATE TRIGGER, RECREATE TRIGGER and CREATE OR ALTER TRIGGER statements.

See also

Trigger

back to top of page

SQL2003 compliant alternative for computed fields

D. Yemanov

Feature request CORE-1386

(**v.2.1**) SQL-compliant alternative syntax GENERATED ALWAYS AS was implemented for defining a computed field in CREATE/ALTER TABLE.

Syntax pattern

```
<column name> [<type>] GENERATED ALWAYS AS ( <expr> )
```

It is fully equivalent semantically with the legacy form:

```
<column name> [<type>] COMPUTED [BY] ( <expr> )
```

Example

```
CREATE TABLE T (PK INT, EXPR GENERATED ALWAYS AS (PK + 1))
```

See also:

New table Alter table

back to top of page

CREATE SEQUENCE

D. Yemanov

SEQUENCE has been introduced as a synonym for GENERATOR, in accordance with SQL-99. SEQUENCE is a syntax term described in the SQL specification, whereas GENERATOR is a legacy InterBase syntax term.

Use of the standard SEQUENCE syntax in your applications is recommended.

A sequence generator is a mechanism for generating successive exact numeric values, one at a time. A sequence generator is a named schema object. In dialect 3 it is a BIGINT, in dialect 1 it is an INTEGER.

Syntax patterns

```
CREATE { SEQUENCE | GENERATOR } <name>
DROP { SEQUENCE | GENERATOR } <name>
SET GENERATOR <name> TO <start_value>
```

```
ALTER SEQUENCE <name> RESTART WITH <start_value>
GEN_ID (<name>, <increment_value>)
NEXT VALUE FOR <name>
```

Examples

1.

```
CREATE SEQUENCE S_EMPLOYEE;
```

2.

```
ALTER SEQUENCE S_EMPLOYEE RESTART WITH 0;
```

See also the notes about NEXT VALUE FOR.

Warning: ALTER SEQUENCE, like SET GENERATOR, is a good way to screw up the generation of key values!

See also:

Generator

back to top of page

REVOKE ADMIN OPTION

D. Yemanov

SYSDBA, the database creator or the owner of an object can grant rights on that object to other users. However, those rights can be made inheritable, too. By using WITH GRANT OPTION, the grantor gives the grantee the right to become a grantor of the same rights in turn. This ability can be removed by the original grantor with REVOKE GRANT OPTION FROM user.

However, there's a second form that involves roles. Instead of specifying the same rights for many users (soon it becomes a maintenance nightmare) you can create a role, assign a package of rights to that role and then grant the role to one or more users. Any change to the role's rights affect all those users.

By using WITH ADMIN OPTION, the grantor (typically the role creator) gives the grantee the right to become a grantor of the same role in turn. Until FB v2, this ability couldn't be removed unless the original grantor fiddled with system tables directly. Now, the ability to grant the role can be removed by the original grantor with REVOKE ADMIN OPTION FROM user.

See also:

- Role
- WITH ADMIN OPTION
- User Manager

- Grant Manager

back to top of page

SET/DROP DEFAULT clauses for ALTER TABLE

C. Valderrama

Domains allow their defaults to be changed or dropped. It seems natural that table fields can be manipulated the same way without going directly to the system tables.

Syntax pattern

```
ALTER TABLE t ALTER [COLUMN] c SET DEFAULT default_value;
ALTER TABLE t ALTER [COLUMN] c DROP DEFAULT;
```

Note:

- Array fields cannot have a default value.
- If you change the type of a field, its default may remain in place. This is because a field could be changed to a defaulted domain, while the field definition itself could override the domain's default. On the other hand, if the field is given a new type directly, any default belongs logically to the field and is maintained on the implicit domain created for it behind the scenes.

See also:

- ALTER TABLE
- Default source

back to top of page

Syntaxes for changing exceptions

D. Yemanov

The DDL statements RECREATE EXCEPTION and CREATE OR ALTER EXCEPTION (feature request SF #1167973) have been implemented, allowing either creating, recreating or altering a custom exception, depending on whether it already exists.

RECREATE EXCEPTION

RECREATE EXCEPTION is exactly like CREATE EXCEPTION if the exception does not already exist. If it does exist, its definition will be completely replaced, if there are no dependencies on it.

CREATE OR ALTER EXCEPTION

CREATE OR ALTER EXCEPTION will create the exception if it does not already exist, or will alter the definition if it does, without affecting dependencies.

See also:

- Exception
- CREATE OR ALTER EXCEPTION

back to top of page

ALTER EXTERNAL FUNCTION

C. Valderrama

ALTER EXTERNAL FUNCTION has been implemented, to enable the entry_point or the module_name to be changed when the UDF declaration cannot be dropped due to existing dependencies.

See also:

User-defined function ALTER EXTERNAL FUNCTION Firebird 2.0.4 Release Notes: External functions (UDFs)

COMMENT statement

C. Valderrama

The COMMENT statement has been implemented for setting metadata descriptions.

Syntax pattern

```
COMMENT ON DATABASE IS {'txt'|NULL};
COMMENT ON <basic_type> name IS {'txt'|NULL};
COMMENT ON COLUMN tblviewname.fieldname IS {'txt'|NULL};
COMMENT ON PARAMETER procname.parname IS {'txt'|NULL};
```

An empty literal string '' will act as NULL since the internal code (DYN in this case) works this way with blobs.

```
<basic_type>:
   DOMAIN
   TABLE
   VIEW
   PROCEDURE
   TRIGGER
   EXTERNAL FUNCTION
   FILTER
   EXCEPTION
   GENERATOR
   SEQUENCE
   INDEX
   ROLE
   CHARACTER SET
   COLLATION
   SECURITY CLASS1
```

1 not implemented, because this type is hidden.

See also:

COMMENT

back to top of page

Extensions to CREATE VIEW specification

D. Yemanov

FIRST/SKIP and ROWS syntaxes and PLAN and ORDER BY clauses can now be used in view specifications.

From Firebird 2.0 onward, views are treated as fully-featured SELECT expressions. Consequently, the clauses FIRST/SKIP, ROWS, UNION, ORDER BY and PLAN are now allowed in views and work as expected.

Syntax

For syntax details, refer to Select Statement & Expression Syntax in the chapter about DML.

RECREATE TRIGGER statement implemented

D. Yemanov

The DDL statement RECREATE TRIGGER statement is now available in DDL. Semantics are the same as for other RECREATE statements.

See also:

Trigger

back to top of page

Usage enhancements

The following changes will affect usage or existing, pre-Firebird 2 workarounds in existing applications or databases to some degree.

Creating foreign key constraints no longer requires exclusive access

V. Horsun

Now it is possible to create foreign key constraints without needing to get an exclusive lock on the whole database.

Changed logic for view updates

Apply NOT NULL constraints to base tables only, ignoring the ones inherited by view columns from domain definitions.

Descriptive identifiers for BLOB subtypes

A. Peshkov, C. Valderrama

Previously, the only allowed syntax for declaring a blob filter was:

```
declare filter <name> input_type <number> output_type <number>
  entry_point <function_in_library> module_name <library_name>;
```

The alternative new syntax is:

```
declare filter <name> input_type <mnemonic> output_type <mnemonic>
  entry_point <function_in_library> module_name <library_name>;
```

where <mnemonic> refers to a subtype identifier known to the engine.

Initially they are binary, text and others mostly for internal usage, but an adventurous user could write a new mnemonic in rdb$types and use it, since it is parsed only at declaration time. The engine keeps the numerical value. Remember, only negative subtype values are meant to be defined by users.

To get the predefined types, do

```
select RDB$TYPE, RDB$TYPE_NAME, RDB$SYSTEM_FLAG
  from rdb$types
  where rdb$field_name = 'RDB$FIELD_SUB_TYPE';
```

RDB$TYPE	RDB$TYPE_NAME RDB$SYSTEM_FLAG	

=========	============================	=================
0	BINARY	1
1	TEXT	1
2	BLR	1
3	ACL	1
4	RANGES	1
5	SUMMARY	1
6	FORMAT	1
7	TRANSACTION_DESCRIPTION	1
8	EXTERNAL_FILE_DESCRIPTION	1

Examples

Original declaration:

```
declare filter pesh input_type 0 output_type 3
  entry_point 'f' module_name 'p';
```

Alternative declaration:

```
declare filter pesh input_type binary output_type acl
  entry_point 'f' module_name 'p';
```

Declaring a name for a user defined blob subtype (remember to commit after the insertion):

```
SQL> insert into rdb$types
CON> values('RDB$FIELD_SUB_TYPE', -100, 'XDR', 'test type', 0);
SQL> commit;
SQL> declare filter pesh2 input_type xdr output_type text
CON> entry_point 'p2' module_name 'p';
SQL> show filter pesh2;
BLOB Filter: PESH2
        Input subtype: -100 Output subtype: 1
        Filter library is p
        Entry point is p2
```