Teil 2: DDL-Trigger

Firebird-3-Vorschau

Der erste Artikel [1] dieser Artikelreihe befasste sich mit ausgewählten Firebird-3-SQL-/PSQL-Spracherweiterungen. In diesem Artikel werden die neu hinzugekommenen DDL-Trigger vorgestellt.

von Thomas Steinmaurer

In Firebird 2.1 wurden spezielle Datenbank-Trigger hinzugefügt, die nicht an Tabellen, sondern vielmehr an datenbankweite Operationen wie zum Beispiel Verbindungsaufbau, Transaktionsstart usw. gebunden sind. Ich berichtete über diesen neuen Trigger-Typ in meinem Artikel über die Neuerungen in Firebird 2.1 [2]. Hat man mit DML-Triggern die Möglichkeit, serverseitigen Code bei *DELETE/INSERT/UPDATE*-Operationen auszuführen, werden Datenbank-Trigger in Firebird 3 mit der Unterstützung für die Ausführung von PSQL-Code bei DDL-Anweisungen in Form von DDL-Triggern erweitert. Es handelt sich hier um Trigger, die bei *CREATE-*, *ALTER-* und *DROP*-Anweisungen angestoßen werden. Folgende Anwendungsszenarien sind für DDL-Trigger denkbar:

- Einhaltung von Namenskonventionen bei der Erstellung von Datenbankobjekten
- Umsetzung eines benutzerdefinierten Sicherheitskonzepts bei der Datenbankobjektanlage
- DDL-Protokollierung für Security Audits, DDL-Replikation usw.

Somit können DDL-Trigger, neben den in Firebird 2.5 eingeführten Audit und Trace Services [3], als zusätzliche Methode für die Protokollierung von ausgeführten DDL-Anweisungen herangezogen werden.

Syntax

Die Syntax zur Erstellung eines DDL-Triggers unterscheidet sich nur unwesentlich von einem DML-Trigger. Einzig und allein das Trigger-Ereignis ist nicht mehr *DELETE/INSERT/UPDATE*, sondern DDL-konforme Ereignisse wie zum Beispiel *CREATE TABLE, CREATE VIEW, ALTER PROCEDURE*. Die vollständige DDL-Trigger-Syntax ist in Listing 1 abgebildet.

Im Wesentlichen stehen auch hier bekannte Konzepte der Trigger-Entwicklung wie *ACTIVE/INACTIVE, BEFORE/AFTER* sowie eine benutzerdefinierte Ausführungsreihenfolge zur Verfügung. Ein DDL-Trigger feuert durch Verwendung der *ANY DDL STATEMENT*-Klausel für eine beliebige DDL-Anweisung oder aber spezialisiert durch Angabe von einem oder mehreren Ereignistypen (*CREATE TABLE, ALTER TABLE* etc.) im Trigger Header.

Kontextinformation

Entscheidend für den Einsatz von Triggern ist die Verfügbarkeit von Kontextinformation. Ist das bei DML-Triggern der Zugriff auf alte/neue Feldwerte über die *OLD.<feld>*- bzw. *NEW.<feld>*-Kontextvariablen, wurde für DDL-Trigger ein neuer Namensraum *DDL_TRIGGER* eingeführt, auf den über die SQL-Funktion *RDB$GET_CONTEXT* zugegriffen werden kann. Die Kontextelemente für diesen neuen Namensraum sind in Tabelle 1 dargestellt.

Will man nun innerhalb eines DDL-Triggers auf den Namen des betroffenen Datenbankobjekts (Tabelle, View usw.) zugreifen, dann sieht der Aufruf von *RDB$GET_CONTEXT* wie folgt aus:

```
RDB$GET_CONTEXT('DDL_TRIGGER', 'OBJECT_NAME')
```

Analog dazu erfolgt der Zugriff zu den anderen DDL-Trigger-spezifischen Kontextinformationen wie dem Ereignistyp, Objekttyp, Ereignisname und der vollständigen DDL-Anweisung. Nicht zu vergessen sind allgemein verfügbare Kontextinformationen über den Namensraum

Kontextelement	Bedeutung
EVENT_TYPE	Ereignistyp: CREATE, ALTER, DROP
OBJECT_TYPE	Objekttyp: TABLE, VIEW
DDL_EVENT	Ereignisname: EVENT_TYPE || ' ' || OBJECT_TYPE (z. B. CREATE TABLE)
OBJECT_NAME	Name der Tabelle, View, Stored Procedure
SQL_TEXT	Vollständige DDL-Anweisung

Tabelle 1: Kontextelemente für DDL_TRIGGER-Namensraum

Listing 1: DDL-Trigger-Syntax

```
<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
```

DDL_LOG_ID	CREATION_TIMESTAMP	USER_NAME	EVENT_TYPE	OBJECT_TYPE	DDL_EVENT	OBJECT_NAME	SQL_TEXT
1	2/21/2011 8:29:20 PM	SYSDBA	CREATE	VIEW	CREATE VIEW	V_T1	create view v_t1 as select * from t1
3	2/22/2011 8:18:44 AM	SYSDBA	CREATE	VIEW	CREATE VIEW	V1_T1	create view v1_t1 as select * From t1
5	2/22/2011 8:20:30 AM	SYSDBA	DROP	VIEW	DROP VIEW	V1_T1	drop view v1_t1
6	2/22/2011 8:21:03 AM	SYSDBA	CREATE	VIEW	CREATE VIEW	V1_T1	create view v1_t1 as select * From t1
7	2/22/2011 8:21:27 AM	SYSDBA	DROP	VIEW	DROP VIEW	V1_T1	drop view v1_t1
8	2/22/2011 8:22:11 AM	SYSDBA	CREATE	VIEW	CREATE VIEW	V1_T1	create view v1_t1 as select * From t1
9	2/22/2011 8:22:30 AM	SYSDBA	DROP	VIEW	DROP VIEW	V1_T1	drop view v1_t1
10	3/2/2011 11:23:27 AM	SYSDBA	CREATE	TABLE	CREATE TABLE	T6	create table t6 (id integer not null primary key)

Abb. 1: Ergebnismenge für erfolgreich ausgeführte DDL-Anweisungen

SYSTEM, wie zum Beispiel die Client-IP-Adresse, das verwendete Netzwerkprotokoll usw., deren Verfügbarkeit nicht auf DDL-Trigger beschränkt ist.

Anwendungsbeispiele

Ein Beispiel je anfangs genanntem Anwendungsszenario soll die Verwendung und den Einsatz von DDL-Triggern erläutern. Im Beispiel aus Listing 2 wird sichergestellt, dass der Name einer zu erstellenden View einer benutzerdefinierten Namenskonvention entspricht. Dazu wird ein Trigger *BEFORE CREATE VIEW* erstellt, der in einer *IF*-Bedingung überprüft, ob der View-Name entsprechend unserer Anforderung mit *V_* beginnt (Listing 2).

Versuchen wir nun, eine View anzulegen, deren Namen nicht unserer Regel entspricht: *create view v1 as select * from t1;* So wird eine Exception ausgelöst, die die Ausführung der *CREATE VIEW*-Anweisung unterbindet und dem Aufrufer die Exception-Meldung von *e_invalid_view_prefix* zurückgibt (Listing 3).

Das zweite Beispiel in Listing 4 zeigt, wie nur ein bestimmter Benutzer Datenbankobjekte erstellen darf. Hier wird die Klausel *ANY DDL STATEMENT* verwendet, damit diese Regel für eine beliebige DDL-Anweisung angewendet wird.

Falls der verbundene Benutzer *DBOWNER* ist, wird die ausgeführte DDL-Anweisung durch eine Exception verworfen und die Exception-Meldung von *e_ddl_*

Listing 2: Beispiel DDL-Trigger – Überprüfung der Namenskonvention

```
recreate exception e_invalid_view_prefix 'Invalid
                                          view prefix.
                  The view name is not starting with V_';

set term !!;
create or alter trigger tri_ddl_view_prefix_check
                        before create view
as
begin
  if (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME')
        not starting with 'V_') then
  begin
    exception e_invalid_view_prefix;
  end
end!!

set term ;!!
commit;
```

Listing 3

```
Statement failed, SQLSTATE = HY000
unsuccessful metadata update
-CREATE VIEW V1 failed
-exception 8
-E_INVALID_VIEW_PREFIX
-Invalid view prefix. The view name is not starting
                                          with V_
```

Listing 4: Benutzerdefiniertes DDL-Sicherheitskonzept

```
recreate exception e_ddl_object_creation_denied
                    'Object creation denied';

set term !!;

create or alter trigger tri_ddl_check_user
                        before any ddl statement
as
begin
  if (current_user <> 'DBOWNER') then
  begin
    exception e_ddl_object_creation_denied;
  end
end!!

set term ;!!
commit;
```

object_creation_denied dem Aufrufer zurückgegeben. Hiermit kann sichergestellt werden, dass zum Beispiel nur der Ersteller der Datenbank (der Datenbankeigentümer) Objekte in der Datenbank erstellen darf.

Das dritte und letzte Beispiel implementiert einen serverseitigen Protokollierungsmechanismus für alle ausgeführten DDL-Anweisungen. Listing 5 zeigt die benötigten Anweisungen für die Erstellung der Logging-Metadaten. Es kommen hier auch SQL-Spracherweiterungen (Auto-Increment, Boolean-Datentyp) aus meinem ersten Artikel [1] dieser Artikelreihe zum Einsatz.

Durch einen Mix aus *BEFORE* und *AFTER ANY DDL STATEMENT* Trigger, in Kombination mit der Verwendung eines *AUTONOMOUS TRANSACTION* Blocks je Trigger, können auch Ausführungsversuche von DDL-Anweisungen mitprotokolliert werden. Schlägt die Ausführung einer DDL-Anweisung fehl, so handelt es sich um einen Ausführungsversuch. Die Logeinträge in der Protokolltabelle *DDL_LOG* sind durch *EXECUTED_SUCCESSFULLY = FALSE* gekennzeichnet. Erfolgreich ausgeführte DDL-Anweisungen werden mit *TRUE* im *AFTER* Trigger über den zwischengespeicherten Primärschlüsselwert des eingefügten Logdatensatzes markiert. Die Logtabelle beinhaltet auch die Werte der Kontextelemente des *DDL_TRIGGER*-Namensraums aus Tabelle 1 und lässt somit keine Wünsche in Bezug auf die Detaildaten der ausgeführten DDL-Anweisung offen. Die Abfrage aus Listing 6 gibt ein DDL-Log der erfolgreich ausgeführten DDL-Anweisungen zurück. Die zurückgegebene Ergebnismenge ist exemplarisch in **Abbildung 1** zu sehen.

Die Anwendungsszenarien für eine umfassende DDL-Protokolltabelle sind vielfältig: Von einfachen „Wer machte was"-Szenarien über gezielte DDL-Inspektionen bis hin zur Replikation von DDL-Anweisungen ist alles denkbar. Der hier gezeigte Protokollierungsmechanismus kann somit als Ausgangsbasis verwendet werden, um beispielsweise eine oder mehrere Produktivdatenbanken mit einer Entwicklungsdatenbank unterschiedlicher Stände abzugleichen. Voraussetzung ist natürlich, dass keine Lücken im DDL-Log entstanden sind. Das kann dadurch entstehen, wenn die benötigten Trigger wieder entfernt bzw. auf inaktiv gesetzt wurden.

Des Weiteren stellt zum Beispiel *isql* die *–nodbtrigger*-Option zur Verfügung, um für eine gestartete *isql*-Session keine Datenbank-Trigger auszuführen. Auch *gbak* und *nbackup* stellen entsprechende Optionen für eine sessionbasierte Deaktivierung von Datenbank-Triggern während einer Sicherung bzw. Wiederherstellung zur Verfügung. Dieses Verhalten ist auch für jede andere Clientanwendung mit der Verwendung des DPB-(Database Parameter-Block-)Werts *isc_dpb_no_db_triggers* zur Verbindungszeit anwendbar. Bei der Verwaltung von Datenbank-Triggern wird allerdings empfohlen, die ses-

sionbasierte Deaktivierung zu verwenden, damit keine Seiteneffekte auftreten. So ein Seiteneffekt könnte zum Beispiel sein, dass die DDL-Protokollierung fehlschlägt, wenn der für die Protokollierung zuständige DDL-Trigger entfernt werden würde. Datenbank-Trigger sind sehr mächtig, allerdings mit Vorsicht zu verwenden.

Ausblick

Mit den neu hinzugekommenen DDL-Triggern schließt das Entwicklerteam eine wichtige Lücke zu kommerziellen Produkten wie Oracle und Microsoft SQL Server. Im Vergleich zu anderen Open-Source-Produkten wie PostgreSQL und MySQL ist das allerdings ein Alleinstellungsmerkmal, da keines der beiden Produkte DDL-Trigger unterstützt.

In einer der nächsten Ausgaben beschäftigen wir uns mit den neuen analytischen SQL-Spracherweiterungen in Firebird 3. Die so genannten *Window Functions* eignen sich besonders für die Beantwortung von analytischen Fragestellungen auf Basis von aggregierten Daten, wie sie vor allem in OLAP/DWH-Umgebungen anzutreffen sind.

Thomas Steinmaurer ist wissenschaftlicher Mitarbeiter am Software Competence Center Hagenberg (SCCH) [4] mit Schwerpunkt Datenmanagement und Data Warehousing im industriellen Umfeld. Des Weiteren ist er für die LogManager-Produktreihe und FB TraceManager bei Upscene Productions [5] verantwortlich.

Links & Literatur

[1] Steinmaurer, Thomas: „Firebird-3-Vorschau – Teil 1“, in Entwickler Magazin 4.2011

[2] Steinmaurer, Thomas: „Firebird 2.1“, in Entwickler Magazin 5.2007

[3] Steinmaurer, Thomas: „Audit und Trace Services in Firebird 2.5“, in Entwickler Magazin 2.2010

[4] http://www.scch.at

[5] http://www.upscene.com

Listing 5: DDL-Protokollierung

```
recreate table ddl_log (
    ddl_log_id bigint generated by default as identity
    , creation_timestamp timestamp default current_timestamp not null
    , user_name varchar(31) default current_user not null
    , event_type varchar(31) not null
    , object_type varchar(31) not null
    , ddl_event varchar(63) not null
    , object_name varchar(31) not null
    , sql_text blob sub_type text not null
    , executed_successfully boolean default true not null
    , constraint pk_ddl_log primary key (ddl_log_id)
);

commit;

set term !!;

create or alter trigger tri_ddl_log_before before any ddl statement
as
  declare vDdlLogId type of column ddl_log.ddl_log_id;
begin
  in autonomous transaction do
  begin
    insert into ddl_log (
      event_type
      , object_type
      , ddl_event
      , object_name
      , sql_text
      , executed_successfully
    )
    values (
      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$get_context('DDL_TRIGGER', 'SQL_TEXT')
      , FALSE
    )
    returning ddl_log_id into :vDdlLogId;
    rdb$set_context('USER_SESSION', 'tri_ddl_log_id', vDdlLogId);
  end
end!!

create or alter trigger tri_ddl_log_after after any ddl statement
as
begin
  in autonomous transaction do
  begin
    update ddl_log set
      executed_successfully = TRUE
    where
      ddl_log_id = rdb$get_context('USER_SESSION', 'tri_ddl_log_id');
  end
end!!

set term ;!!

commit;
```

Listing 6: SQL-Abfrage zur Ermittlung erfolgreich ausgeführter DDL-Anweisungen

```
select
  ddl_log_id
  , creation_timestamp
  , user_name
  , event_type
  , object_type
  , ddl_event
  , object_name
  , sql_text
from
  ddl_log
where
  executed_successfully
order by
  ddl_log_id
```