Die ersten SQL/PSQL-Erweiterungen im Überblick – Teil 1

Firebird-3-Vorschau

Dass nach der Veröffentlichung von Firebird 2.5 keine Ruhe in die Entwicklung eingekehrt ist, zeigen die unzähligen Change-Log-Einträge im Firebird-Hauptentwicklungszweig, der die Entwicklungen für Firebird 3 widerspiegelt. Dieser Artikel ist der erste Teil einer Artikelreihe, die einen Blick hinter die Kulissen der bereits umgesetzten Neuerungen für das nächste Major Release wagt.

von Thomas Steinmaurer

Das finale Release von Firebird 2.5 wurde im Oktober 2010 präsentiert [1]. Mit einer für dieses Release speziell initiierten Releasekampagne [2] hat man versucht, diese Version der Öffentlichkeit schmackhaft zu machen. Wo man mit neuen Projekten bereits auf den „2.5-Zug“ aufgesprungen ist, werden Legacy-Umgebungen langsam, aber stetig auf Version 2.5 umgestellt. Zu gravierend sind die Neuerungen, um ein Update nicht in Betracht zu ziehen. Vor allem die verbesserte Unterstützung von Mehrprozessorsystemen und die Möglichkeit eines kontinuierlichen Monitorings mit dem Trace API sind für einen Produktiveinsatz nicht mehr wegzudenken.

Snapshot Builds

Obwohl mit Stand Mai 2011 noch keine erste freigegebene Alphaversion von Firebird 3 zur Verfügung stand, ermöglichen die verfügbaren Snapshot Builds [3], einen ersten Blick auf die kommenden Neuerungen zu werfen. Dies ist eine Stärke eines Open-Source-Projekts, und das Entwicklerteam erwartet bereits in diesem frühen Stadium konstruktives Feedback. Die verfügbaren Binaries sind bewusst als „unstable“ gekennzeichnet, da sich während der Entwicklung noch vieles wird ändern können. Man nimmt hier keine Rücksicht auf die Abwärtskompatibilität zu älteren V3 Snapshot Builds. Im schlimmsten Fall ist eine erstellte Datenbank mit einem neueren Snapshot Build nicht mehr verwendbar. Somit ist Vorsicht geboten, da man sich hier ganz klar auf einer Spielwiese für Enthusiasten befindet. Folgende Neuerungen aus dem Firebird-3-Entwicklungszweig werden hier vorgestellt:

- *Boolean*-Datentyp: spezieller Datentyp für boolesche Werte
- *Identity*-Spalten: vereinfachte Erstellung von Auto-Increment-Spalten, z. B. für Primärschlüsselfelder

- *Stored Functions*: serverseitige Codemodule ähnlich zu Stored Procedures, jedoch mit dem Unterschied, dass die Rückgabe gemäß einer Funktion ein skalarer Wert ist
- *Packages*: neuer Datenbankobjekttyp, der ähnlich zu Oracle die logische Gruppierung von gespeicherten Prozeduren und Funktionen ermöglicht

Ebenfalls bereits umgesetzt sind *DDL Trigger* und die so genannten *Window Functions*. Beides wird in den nächsten Ausgaben des Entwickler Magazins behandelt. In diesem Artikel gehe ich bewusst auch nicht auf die Vereinheitlichung der Serverarchitekturen und der geplanten Verbesserungen im Bereich des Multithreadings ein. Das ist zum jetzigen Zeitpunkt noch zu früh. Auch die angekündigte Unterstützung für Java-basierte Stored Procedures/Functions ist mit Stand Mai 2011 in den Snapshots noch nicht verfügbar.

Boolean-Datentyp

Mit Firebird 3 wird ein nativer *Boolean*-Datentyp zur Verfügung stehen. Somit sind Kniffe und deren Nachteile mit einer eigenen Domain (z. B. *CHAR(1)*) nicht mehr notwendig. Nachfolgende Anweisung *CREATE TABLE* inkludiert eine Spalte vom Typ *BOOLEAN*.

```
create table t1 (
  t1_id integer not null
  , b boolean default true
);
commit;
```

Als Wertausprägungen können *TRUE* und *FALSE* verwendet werden, aber natürlich auch *NULL* bzw. das neu eingeführte Schlüsselwort *UNKNOWN* für einen unbekannten Zustand. Sehen wir uns in weiterer Folge ein paar Anwendungsbeispiele an. Die Anweisungen *EXECUTE BLOCK* und *UPDATE* aus Lising 1 fügen 1 000 Datensätze ein und setzen für jeden zweiten Datensatz das Feld *b* auf den Wert *FALSE*.

Für den Zugriff auf die Datensätze mit *b* = *TRUE* hat man nun unterschiedliche Möglichkeiten:

```
select count(*) from t1 where b = true;
select count(*) from t1 where b is true;
select count(*) from t1 where b;
```

Da kein Index auf der Spalte *b* existiert, wird in allen drei Fällen ein Full-Table Scan (*PLAN (T1 NATURAL)*) durchgeführt. Beim Vorhandensein eines Index der Form:

```
create index i_t1_b on t1 (b);
commit;
```

wird dieser für den Zugriff auch verwendet (*PLAN (T1 INDEX (I_T1_B))*). Der Zugriff über den Index funktioniert auch mit der Negation auf *b* bzw. *TRUE*.

```
select count(*) from t1 where b <> true;
select count(*) from t1 where b is not true;
select count(*) from t1 where not b;
```

Dies ist bei der Verwendung einer Domain mit z. B. *CHAR(1)* nicht möglich. In allen Fällen ist das Ergebnis:

```
      COUNT
============
         500
```

Identity-Spalten

In Firebird 3 wird es erstmals möglich sein, eine Auto-Increment-Spalte zu erstellen, ohne dabei selbst den dazu benötigten Generator und *BEFORE INSERT* Trigger erstellen zu müssen. War dies für Firebird-Einsteiger/Umsteiger noch verwirrend bzw. eine erste gute Übung im Umgang mit Generatoren und Triggern, ist das in Firebird 3 nicht mehr notwendig (Listing 2).

Die implementierte syntaktische Erweiterung ist grundsätzlich konform zum SQL2003-Standard, allerdings mit ein paar Abstrichen. So fehlt die Unterstützung der *ALWAYS*-Klausel, die anstatt von *BY DEFAULT* verwendet werden kann. Mit der *ALWAYS*-Klausel würden benutzer-

Listing 1: Execute Block und Update mit Boolean-Spalte

```
set term ## ;
execute block
as
declare i Integer;
begin
  i = 1;
  while (i <= 1000) do
  begin
    insert into t1 (t1_id) values (:i);
    i = i + 1;
  end
end ##
set term ; ##
update t1 set b = false where mod(t1_id, 2) = 0;
commit;
```

Listing 2

```
create table t2 (
  t2_id bigint generated by default as identity
  , i integer
  , constraint pk_t2 primary key (t2_id)
);
commit;
```

Listing 3

```
select
  rdb$generator_name
  , rdb$identity_type
from
  rdb$relation_fields
where
  rdb$relation_name = 'T2'
  and rdb$field_name = 'T2_ID';
```

Ergebnis:

```
RDB$GENERATOR_NAME          RDB$IDENTITY_TYPE
==================================================
RDB$2                       1
```

definierte Werte, die für eine Identity-Spalte in einer DML-Anweisung angegeben werden, ignoriert werden.

Um die dazu benötigten Metadaten zu speichern, wurde die Systemtabelle *RDB$RELATION_FIELDS* um die Felder *RDB$GENERATOR_NAME* und *RDB$IDENTITY_TYPE* erweitert. Für besonders Neugierige kann diese Information mit der folgenden Systemtabellenabfrage in Listing 3 abgefragt werden.

Der von der Firebird Engine erstellte Generator *RDB$2* ist als solcher auch in der Systemtabelle *RDB$GENERATORS* ersichtlich. Mit einem Wert *RDB$SYSTEM_FLAG = 6* ist speziell gekennzeichnet, dass es sich um einen systemgenerierten Identity-Generator handelt. Nun könnte man auf die Idee kommen, den Generator *RDB$2* mit folgender DDL-Anweisung zu entfernen: *„drop generator rdb$2;“*. Was allerdings mit der folgenden Exception unterbunden wird:

```
Statement failed, SQLSTATE = 42000
unsuccessful metadata update
-Cannot delete system generator
```

Wie können nun Identity-Spalten in DML-Anweisungen verwendet werden? Das zeigt das Beispiel in Listing 4. Nur die dritte Anweisung führt explizit einen Wert für *t2_id* an. Im Fall der Verwendung der Klausel *BY DEFAULT* bei der Spaltendeklaration wird dieser anstatt des nächsten Generatorwerts auch verwendet. Allerdings resultiert das vierte *INSERT INTO* nicht in einem Wert 101 für *t2_id*, wie das zum Beispiel bei SQLite der Fall wäre, sondern es wird 3 abgespeichert. Ein *SELECT ** auf die Tabelle zeigt ein Ergebnis wie in Listing 5.

Das birgt natürlich die Gefahr, dass es zu einer Primärschlüsselverletzung kommt, wenn der Generatorwert die 100 erreicht. Man sollte hier also mit der manuellen Zuweisung von Primärschlüsselwerten vorsichtig sein. Leider unterstützt Firebird 3 die *ALWAYS*-Klausel nicht, die benutzerdefinierte Werte ignorieren würde. Vielleicht legt hier das Entwicklerteam noch nach, um dem SQL-Standard noch näher zu kommen.

Bei dem im Hintergrund erzeugten Systemgenerator handelt es sich um einen üblichen Generator, auf den mit *GEN_ID(<generator name>, <inkrement>)* zugegriffen werden kann. Für die Ermittlung des generierten Werts im Zuge einer Einfüge-Operation sollte man allerdings auf die *RETURNING*-Klausel einer DML-Anweisung zurückgreifen, da ein *GEN_ID (RDB$2, 0)* in einer Mehrbenutzerumgebung nicht zuverlässig wäre. Zum Beispiel *insert into t2 (i) values (1004) returning t2_id;* ergibt folgende Ausgabe in *isql*:

```
              T2_ID
====================
                  4
```

Kombiniert mit der Verwendung in PSQL kann der zurückgegebene Primärschlüsselwert in einer lokalen Variable gespeichert und dann weiter verarbeitet werden.

Stored Functions

Stehen die serverseitigen Programmiertechniken *Stored Procedures* und *Trigger* seit Beginn von Firebird zur Verfügung, werden in Firebird 3 auch *Stored Functions* möglich sein. Dabei handelt es sich, analog zu Stored Procedures, um serverseitige Codemodule, die allerdings

Listing 4

```
insert into t2 (i) values (1000);
insert into t2 (i) values (1001);
insert into t2 (t2_id, i) values (100, 1002);
insert into t2 (i) values (1003);
commit;
```

Listing 5

```
              T2_ID            I
==================== ============
                   1         1000
                   2         1001
                 100         1002
                   3         1003
```

Listing 6: Stored Function

```
set term ##;
create function foo (i integer) returns integer
as
begin
  return i * 3;
end ##
set term ;##

commit;
select
  foo(2)
from
  rdb$database;
```

Listing 7: Package

```
set term !!;

create or alter package pkg_schema
as
begin
  procedure s_user_tables returns (
    relation_name type of column rdb$relations
                                 .rdb$relation_name
  );
end!!

recreate package body pkg_schema
as
begin
  procedure s_user_tables returns (
    relation_name type of column rdb$relations.
                                 rdb$relation_name
  )
  as
  begin
    for
      select
        rdb$relation_name
      from
        rdb$relations
      where
        (rdb$system_flag = 0 or rdb$system_flag is null)
        and (rdb$view_blr is null)
      order by
        rdb$relation_name asc
      into
        :relation_name
    do
    begin
      suspend;
    end
  end;
end!!

set term ;!!

commit;
```

als Ergebnis einen skalaren Wert zurückliefern. War dies bereits in einer ähnlicher Form mit Stored Procedures möglich – wobei hier die Verwendung alles andere als intuitiv war – steht auch mit Stored Functions nun der vollständige PSQL-Umfang für die Entwicklung serverseitiger und gespeicherter Funktionen zur Verfügung. Harte Konkurrenz für die teilweise ungeliebten User Defined Functions (UDF), die sich auf die Serverstabilität negativ auswirken können, falls die UDF in Bezug auf Speicherverwaltung und Multithreading nicht ordentlich implementiert sind. Aber vor allem erschweren UDFs die Portabilität der Datenbank auf unterschiedliche Plattformen. Eine UDF-Bibliothek muss immer für die Zielplattform (Betriebssystem und 32/64-bit) übersetzt werden. Probleme, die auf Stored Functions nicht zutreffen. Diese sind somit eine sehr gelungene Erweiterung für jeden PSQL-Entwickler. Ein sehr einfaches Beispiel für das Erstellen einer Stored Function und deren Anwendung in einer *SELECT*-Anweisung zeigt das Beispiel aus Listing 6. Das Ergebnis ist keine große Überraschung:

```
          FOO
============
            6
```

Wichtig ist zu erkennen, dass die Verwendung von PSQL in Form einer Funktion nun intuitiv möglich ist. Die Speicherung der Metadaten für die Stored Functions erfolgt in der bereits vorhandenen Systemtabelle *RDB$FUNCTIONS*, die in vorangegangen Firebird-Versionen ausschließlich für die Ablage von UDF-Definitionen verwendet wurde. Diese Systemtabelle wurde mit zusätzlichen Feldern (z. B.: *RDB$FUNCTION_SOURCE* usw.) erweitert, um auch Stored Functions darin verwalten zu können.

Packages

Stored Functions sind nur eine Erweiterung im Kontext von PSQL. In Firebird 3 geht es der Organisation von Stored Procedures/Functions an den Kragen. Es wird das Konzept der *Packages*, wie man sie z. B. aus Oracle kennt, zur Verfügung stehen. Eine immens wichtige Neuerung, um bei sehr Stored-Procedures/Functions-lastigen Datenbanken eine benutzerdefinierte, logische Ordnung definieren zu können. Im Wesentlichen besteht ein Package aus einem *Header* und einem *Body*. Der Package Header definiert die Schnittstelle, und der Package Body beinhaltet die Implementierung der Schnittstelle (Listing 7).

Die Absicht hinter der Erstellung des Package *pkg_schema* ist die Kapselung eines vereinfachten Zugriffs auf die Firebird-Systemtabellen (*RDB$*). Das Package ist natürlich ausbaufähig, denn die einzige implementierte Stored Procedure gibt die verfügbaren Benutzertabellen zurück. Die Verwendung einer Stored Procedure/Function aus einem Package folgt der Punktnotation und sieht dann wie folgt aus:

```
select * from pkg_schema.s_user_tables;
```

In Bezug auf die Zugriffsberechtigung müssen für *SYSDBA* und den Package-Eigentümer keine weiteren Aktionen getroffen werden. Anderen Benutzern bzw. Rollen muss explizit der Zugriff auf Package-Ebene gewährt werden. Das erfolgt mit einer regulären *GRANT*-Anweisung. Ein Beispiel auf Benutzerebene: *grant execute on package pkg_schema to user u1;* bzw. auf Rollenebene: *grant execute on package pkg_schema to role r1;*.

Mitglieder (Benutzer) der Rolle *r1* haben somit Zugriff auf das Package und deren Stored Procedures/Functions, sofern zum Zeitpunkt der Datenbankverbindung die Rolle *r1* mit angegeben wurde.

Für die Speicherung der Package-Metadaten ist eine neue Systemtabelle *RDB$PACKAGES* hinzugekommen bzw. wurde *RDB$PROCEDURES* unter anderem mit einem neuen Feld *RDB$PACKAGE_NAME* erweitert. Ein interessantes Implementierungsdetail ist, dass für Stored Procedures/Functions, die sich in einem Package befinden, nur mehr die kompilierte Form (BLR-Code) in *RDB$PROCEDURES* bzw. *RDB$FUNCTIONS* gespeichert ist. Der Sourcecode ist rein in *RDB$PACKAGES* abgelegt. Somit werden Redundanzen vermieden.

Fernab von Packages möchte ich mit dem im Package verwendeten PSQL-Code noch auf eine wertvolle Neuerung aus Firebird 2.5 eingehen, nämlich das Konstrukt *TYPE OF COLUMN* in der Parameterliste, das es ermöglicht, einen Datentyp basierend auf einer vorhandenen Tabellenspalte mehrmals zu verwenden. Sollte zum Beispiel in naher Zukunft die maximale Länge von Datenbankobjektnamen, in diesem speziellen Fall von Tabellennamen, erhöht werden, dann ist keine Änderung im PSQL-Code notwendig.

Ausblick

Die hier vorgestellten SQL/PSQL-Spracherweiterungen sind nur der Anfang. *DDL Trigger* und *Window Functions* werden in den nächsten Teilen dieser Artikelreihe diskutiert. Gespannt darf man auch auf die Verschmelzung der aktuell vorhandenen Serverarchitekturen *ClassicServer*, *SuperServer*, *SuperClassic* sein. Weitere Verbesserungen in der Unterstützung von Mehrprozessorsystemen sind ebenfalls angekündigt. Ich werde Sie auf dem Laufenden halten.

Thomas Steinmaurer ist wissenschaftlicher Mitarbeiter am Software Competence Center Hagenberg (http://www.scch.at) mit den Schwerpunkten Datenmanagement und Data Warehousing im industriellen Umfeld. Des Weiteren ist er für die LogManager-Produktreihe und FB TraceManager bei Upscene Productions (http://www.upscene.com) verantwortlich.

Links & Literatur

[1] http://www.firebirdsql.org/pop/pop_pressRelease25_de.html

[2] http://www.mindthebird.com/

[3] http://firebirdsql.org/index.php?op=files&id=snapshots