Firebird Generatoren-Ratgeber

Ein Ratgeber über Art und Zweck der Anwendung von Generatoren in Firebird

Frank Ingermann

4. Dezember 2006 – Dokumentenversion 0.2-de

Übersetzung ins Deutsche: Frank Ingermann

Inhaltsverzeichnis

Einführung

Wovon handelt dieser Artikel?

Dieser Artikel erklärt, was Firebird-Generatoren sind, und wie und warum man sie benutzen kann. Dies ist der Versuch, alle relevanten Informationen über Generatoren in einem Dokument zusammenzufassen.

Wer sollte ihn lesen?

Lese diesen Artikel, wenn...

- dir das Konzept von Generatoren neu ist;
- du Fragen zu ihrer Anwendung hast;
- du eine Integer-Spalte so verwenden willst wie ein "AutoInc"-Feld, wie man es aus anderen RDBMS kennt
- du Beispiele zur Anwendung von Generatoren für IDs und andere Zwecke suchst;
- du wissen möchtest, wie die Sequenzen aus Oracle in Firebird heißen

Generatoren: Grundlagen

Was ist ein Generator?

Man stelle sich einen Generator vor wie einen "Thread-sicheren" Integer-Zähler, der in einer Firebird Datenbank existiert. Man kann einen Generator erzeugen, in dem man ihm einen Namen gibt:

```
CREATE GENERATOR GenTest;
```

Dann kann man seinen Wert abfragen, erhöhen oder verringern, so wie man es mit einem "var i:Integer" in Delphi tun kann. Es ist aber nicht immer einfach, ihn zuverlässig auf einen bestimmten Wert zu setzen und diesen dann abzufragen - er ist in der Datenbank, aber *ausserhalb jeder Transaktionskontrolle.*

Was ist eine Sequenz?

„Sequence“ ist der offizielle SQL-Begriff für das, was Firebird einen Generator nennt. Da Firebird ständig in Richtung besserer Unterstützung des SQL-Standards entwickelt wird, kann man von Firebird 2 an den Begriff SEQUENCE als Synonym für GENERATOR benutzen. Es wird in der Tat empfohlen, in neuem SQL-Code die SEQUENCE-Syntax zu verwenden.

Auch wenn das Schlüsselwort "SEQUENCE" die Betonung auf die Erzeugung einer Serie von Werten legt, während GENERATOR eher auf die "Fabrik" zur Erzeugung der Werte zu deuten scheint, gibt es *keinerlei Unterschied* in Firebird zwischen einer SEQUENCE und einem GENERATOR. Es sind einfach zwei Namen für das gleiche Datenbankobjekt. Man kann einen Generator erzeugen und dann mit der Sequenz-Syntax auf ihn zugreifen und umgekehrt.

Dies ist die bevorzugte Syntax zur Erzeugung einer Sequenz bzw. eines Generators in Firebird 2:

```
CREATE SEQUENCE SeqTest;
```

Wo werden Generatoren gespeichert?

Die *Deklarationen* von Generatoren werden in der Systemtabelle RDB$GENERATORS abgelegt. Ihre aktuellen *Werte* hingegen liegen in speziell reservierten Seiten (Pages) in der Datenbank. Man bearbeitet diese Werte nie direkt, sondern durch eingebaute Funktionen und Befehle, die im Verlauf dieses Artikels besprochen werden.

> **Warnung**
>
> Die Informationen in diesem Abschnitt sind nur zu Lernzwecken angegeben. Grundsätzlich sollte man nie direkt mit den Systemtabellen arbeiten. Versuche nicht, Generatoren zu erzeugen oder zu ändern, in dem du direkt die Systemtabelle RDB$GENERATORS manipulierst. (Ein SELECT kann allerdings nicht schaden).

Die Struktur der RDB$GENERATORS Systemtabelle sieht wie folgt aus:

- RDB$GENERATOR_NAME CHAR(31)
- RDB$GENERATOR_ID SMALLINT
- RDB$SYSTEM_FLAG SMALLINT

Und, von Firebird 2.0 an:

- RDB$DESCRIPTION BLOB subtype TEXT

Man beachte, dass die GENERATOR_ID – wie der Name schon sagt – ein IDentifizierer für jeden Generator ist, und *nicht* sein Wert. Man sollte auch nicht diese ID in seinen Anwendungen benutzen, um später auf Generatoren zuzugreifen. Abgesehen davon, dass dies wenig Sinn macht (der *Name* identifiziert den Generator), kann sich die GENERATOR_ID nach einem Backup und anschliessendem RESTORE ändern. Das SYSTEM_FLAG ist 1 für die in der Datenbank intern verwendeten Generatoren, und NULL oder 0 für alle selbsterzeugten.

Werfen wir einen Blick auf die RDB$GENERATORS-Tabelle, hier mit einem einzigen selbstdefinierten Generator:

RDB$GENERATOR_NAME	RDB$GENERATOR_ID	RDB$SYSTEM_FLAG
RDB$SECURITY_CLASS	1	1
SQL$DEFAULT	2	1
RDB$PROCEDURES	3	1
RDB$EXCEPTIONS	4	1
RDB$CONSTRAINT_NAME	5	1

RDB$GENERATOR_NAME	RDB$GENERATOR_ID	RDB$SYSTEM_FLAG
RDB$FIELD_NAME	6	1
RDB$INDEX_NAME	7	1
RDB$TRIGGER_NAME	8	1
MY_OWN_GENERATOR	9	NULL

Firebird 2 Anmerkungen

- Firebird 2 hat einen neuen System-Generator eingeführt namens RDB$BACKUP_HISTORY. Dieser wird vom neuen NBackup-Feature verwendet.
- Auch wenn die SEQUENCE-Syntax jetzt bevorzugt wird, wurden die RDB$GENERATORS Systemtabelle und ihre Spalten in Firebird 2 nicht umbenannt.

Was ist der maximale Wert eines Generators?

Generatoren speichern und liefern 64-bit Integerwerte in allen Firebird-Versionen. Dies ergibt einen Wertebereich von:

-2^{63} .. 2^{63}-1 oder -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

Würde man also einen Generator mit Startwert 0 benutzen, um damit eine NUMERIC(18) oder BIGINT-Spalte zu befüllen, und man würde 1000 neue Datensätze pro Sekunde anlegen, dann würde es etwa 300 Millionen Jahre (!) dauern bevor der Generator überläuft. Da es eher unwahrscheinlich ist, dass die Menschheit dann noch auf diesem Planeten herumläuft (und immer noch Firebird-Datenbanken einsetzt), braucht man sich darüber also nicht wirklich Gedanken machen.

Hier aber ein Wort der Warnung: Firebird spricht zwei SQL-"Dialekte": Dialekt 1 und Dialekt 3. Neue Datenbanken sollten immer mit dem in vieler Hinsicht mächtigeren Dialekt 3 erstellt werden. Dialekt 1 dient nur der Abwärtskompatibilität für Datenbanken, die mit InterBase 5.6 und früheren Versionen erstellt wurden.

Einer der Unterschiede zwischen den beiden liegt darin, dass Dialekt 1 keinen nativen 64bit-Integer-Typen kennt. NUMERIC(18)-Spalten beispielsweise werden intern als DOUBLE PRECISION abgespeichert, was aber ein Gleitkommawert ist. Der größte verfügbare Integer-Typ in Dialekt 1 ist der 32bit-Integer.

In Dialekt 1 wie auch in Dialekt 3 haben Generatoren 64bit. Wenn man aber einen Generatorwert in einer Dialekt 1-Datenbank einer INTEGER-Spalte zuweist, werden die oberen 32bit abgeschnitten, so dass man einen effektiven Wertebereich erhält von:

-2^{31} .. 2^{31}-1 oder -2,147,483,648 .. 2,147,483,647

Auch wenn der Generator selbst von 2,147,483,647 zu 2,147,483,648 und weiterläuft, würde der abgeschnittene Wert in der Spalte an dieser Stelle überlaufen und den *Eindruck* eines 32bit-Generators erwecken.

In der oben beschriebenen Situation mit 1000 Datensätzen pro Sekunde würde die vom Generator gefüllte Spalte nun nach 25 *Tagen* (!!!) überlaufen, und dem sollte auf jeden Fall Beachtung geschenkt werden. 2^{31} ist eine ganze Menge, aber je nach Situation auch wieder nicht so viel.

> **Anmerkung**
>
> In Dialekt 3 geht die Zuweisung von Generator-Werten an INTEGER-Spalten solange gut, wie der Wert im 32bit-Integer-Bereich liegt. Sobald aber dieser Bereich überschritten wird, gibt es einen Numerischen Überlaufsfehler ("numeric overflow error"): Dialekt 3 ist viel strikter in der Bereichsüberprüfung als Dialekt 1!

Client-Dialekte und Generatorwerte

Clients, die mit einem Firebird-Server verbunden sind, können ihren Dialekt auf 1 oder 3 stellen, und zwar unabhängig von der verbundenen Datenbank. Es ist der Dialekt des Clients, *nicht* der der Datenbank, der entscheidet, wie Firebird Generatorwerte zum Client liefert:

- Wenn der Client-Dialekt 1 ist, liefert der Server Generatorwerte als abgeschnittene 32bit-Werte zum Client. Aber innerhalb der Datenbank bleiben sie 64bit-Werte und laufen nach Erreichen von 2^{31}-1 nicht über (auch wenn das für den Client so aussieht). Dies gilt sowohl für Dialekt 1 wie für Dialekt 3-Datenbanken.
- Wenn der Client-Dialekt 3 ist, gibt der Server volle 64 Bit zum Client zurück. Auch dies gilt für beide Datenbank-Dialekte.

Wie viele Generatoren sind in einer Datenbank verfügbar?

Seit Firebird 1.0 ist die Anzahl der verfügbaren Generatoren nur durch den größtmöglichen Wert für die ID-Spalte in der RDB$GENERATORS-Systemtabelle limitiert. Da dies eine SMALLINT-Spalte ist, ist die max. Anzahl 2^{15}-1 oder 32767. Die erste ID ist immer 1, d.h. die Gesamtanzahl der Generatoren kann 32767 nicht überschreiten. Wie zuvor beschrieben, gibt es in jeder Datenbank 8 oder 9 Systemgeneratoren, so dass effektiv noch mindestens 32758 für eigene Generatoren übrig bleiben. Dies sollte für jede praktische Anwendung bei weitem ausreichen. Da die Anzahl der Generatoren keine Auswirkung auf die Performanz hat, kann man nach Herzenslust so viele Generatoren benutzen wie man möchte.

Ältere InterBase- und Firebird-Versionen

In den frühesten vor-1.0 Firebird-Versionen, so wie in InterBase, wurde nur eine Datenbankseite (Page) zur Speicherung der Generatorwerte benutzt. Dadurch war die Anzahl nutzbarer Generatoren durch die Seitengröße (Page Size) der Datenbank begrenzt. Die folgende Tabelle zeigt, wie viele Generatoren (inkl. der Systemgeneratoren) in den verschiedenen InterBase- und Firebird-Versionen zur Verfügung stehen (mit Dank an Paul Reeves für diese Informationen):

Version	**Seitengrösse (Page size)**			
	1K	*2K*	*4K*	*8K*
InterBase < v.6	247	503	1015	2039
IB 6 und frühe Prä-1.0 Firebird	123	251	507	1019
Alle späteren Firebird-Vers.	32767			

In InterBase-Versionen vor 6 waren Generatoren nur 32 Bit breit. Dies erklärt, warum diese früheren Versionen ungefähr die doppelte Anzahl an Generatoren in der selben Seitengrösse speichern konnten.

> **Warnung**
>
> InterBase, zumindest bis inklusive Version 6.01, ließ problemlos die Erzeugung von bis zu 32767 Generatoren zu. Was passierte, wenn man auf Generatoren mit einer ID grösser der oben angegeben Maximalzahl zugriff, hing von der Version ab:
>
> - InterBase 6 generierte einen „invalid block type“-Fehler da die berechnete Position ausserhalb der einen reservierten Generatoren-Seite lag.
> - In früheren Versionen wurde ein Fehler gemeldet, wenn die berechnete Position ausserhalb der Datenbank lag. Ansonsten wurde beim *Lesezugriff* einfach der Wert geliefert, der sich zufällig an der berechneten Position befand. Wurde der "zu grosse" Generator verändert, dann *überschrieb* er einfach den Wert an der berechneten Position. Manchmal führte dies zu einem sofortigen Fehler, meistens aber einfach zu einer stillen Beschädigung der Datenbank.

Generatoren und Transaktionen

Wie gesagt leben Generatoren ausserhalb der Transaktionskontrolle. Dies bedeutet schlicht und ergreifend, dass es keinen sicheren Weg gibt, in einer Transaktion ein "Rollback" eines Generators durchzuführen. Andere, zeitgleich laufende Transkationen können den Wert verändern, während die eigene Transaktion läuft. Hat man also einen Generatorwert erzeugt, sollte man ihn als "auf ewig verbraucht" betrachten.

Startet man also eine Transaktion und erzeugt darin einen Generatorwert von - sagen wir - 5, dann bleibt der Generator auf diesem Wert, **selbst wenn man ein Rollback der Transaktion durchführt (!)**. Man sollte nicht mal *denken* an etwas wie: "OK, wenn ich ein Rollback durchführe, setze ich den Generator mittels GEN_ID(mygen,-1) eben wieder auf 4 zurück". Dies kann meistens funktionieren, ist aber *unsicher*, da andere Transaktionen den Wert inzwischen wiederum verändert haben können. Aus dem gleichen Grund macht es keinen Sinn, den aktuellen Generatorwert mit GEN_ID(mygen,0) aus der Datenbank zu holen und ihn dann Client-seitig zu inkrementieren.

SQL-Befehle für Generatoren

Befehlsüberblick

Der Name des Generators muss ein üblicher Bezeichner für DB-Objekte sein: 31 Zeichen Maximallänge, keine Sonderzeichen mit Ausnahme des Unterstrichs „_“ (es sei denn, man verwendet "delimited identifier", d.h. Bezeichner in Anführungsstrichen). Die SQL-Befehle für Generatoren sind unten aufgeführt. Ihre Verwendung wird detailliert im Abschnitt *Verwendung der Generator-Befehle* beschrieben.

DDL (Data Definition Language) - Befehle:

```
CREATE GENERATOR <name>;
SET GENERATOR <name> TO <value>;
```

```
DROP GENERATOR <name>;
```

DML (Data Manipulation Language) Befehle in Client-seitigem SQL:

```
SELECT GEN_ID( <GeneratorName>, <increment> ) FROM RDB$DATABASE;
```

DML Anweisungen in PSQL (Procedural SQL, verfügbar in Stored Procedures und Triggern):

```
<intvar> = GEN_ID( <GeneratorName>, <increment> );
```

Für Firebird 2 empfohlene Syntax

Auch wenn die traditionelle Syntax weiter unterstützt wird, sind dies die für Firebird 2 bevorzugten Äquivalente der DDL-Befehle:

```
CREATE SEQUENCE <name>;
ALTER SEQUENCE <name> RESTART WITH <value>;
DROP SEQUENCE <name>;
```

Und für die DML-Befehle:

```
SELECT NEXT VALUE FOR <SequenceName> FROM RDB$DATABASE;
```

```
<intvar> = NEXT VALUE FOR <SequenceName>;
```

Derzeit unterstützt die neue Syntax ausschliesslich ein Inkrement von 1. Diese Einschränkung wird in einer zukünftigen Version aufgehoben. In der Zwischenzeit kann man die GEN_ID-Syntax nutzen, falls man einen anderen Inkrement benötigt.

Verwendung der Generator-Befehle

Die Verfügbarkeit der Befehle und Funktionen hängt davon ab, wo man sie benutzt:

- In Client-seitigem SQL – die Sprache, in der der Client mit dem Firebird-Server kommuniziert.
- PSQL – Die Server-seitige Programmiersprache, die in Stored Procedures und Triggern verwendet wird.

Einen Generator erzeugen („Insert“)

Client SQL

```
CREATE GENERATOR <GeneratorName>;
```

Für Firebird 2 und höher bevorzugt:

```
CREATE SEQUENCE <SequenceName>;
```

PSQL

Nicht möglich. Da man die Metadaten der Datanbank innerhalb von SPs und Triggern nicht ändern kann, kann man hier auch keine Generatoren erzeugen

> **Anmerkung**
>
> Seit FB 1.5 und aufwärts kann man dies durch die Verwendung des EXECUTE STATEMENT-Features umgehen.

Den aktuellen Wert abfragen („Select“)

Client SQL

```
SELECT GEN_ID( <GeneratorName>, 0 ) FROM RDB$DATABASE;
```

Diese Syntax ist auch in Firebird 2 derzeit die einzige Option.

> **Anmerkung**
>
> In Firebird's Kommandozeilen-Werkzeug *isql* gibt es zwei weitere Befehle zum Auslesen der aktuellen Generatorwerte::
>
> ```
> SHOW GENERATOR <GeneratorName>;
> SHOW GENERATORS;
> ```
>
> Der erstere zeigt den aktuellen Wert des angegeben Generators, letzterer tut dies für alle Nicht-System-Generatoren in der Datenbank.
>
> Die für Firebird 2 bevorzugten Äquivalente sind, man ahnt es schon:
>
> ```
> SHOW SEQUENCE <SequenceName>;
> SHOW SEQUENCES;
> ```
>
> Nochmals der Hinweis: Diese SHOW-Befehle stehen nur in isql zur Verfügung. Anders als GEN_ID, können sie nicht in Client-seitigem SQL verwendet werden (es sei denn, die Client-Anwendung ist eine isql-Oberfläche oder Frontend).

PSQL

```
<intvar> = GEN_ID( <GeneratorName>, 0 );
```

Firebird 2: Gleiche Syntax.

Den nächsten Wert generieren („Update“ + „Select“)

Genau wie das Ermitteln des aktuellen Werts wird dies mittels GEN_ID errecicht, diesmal aber mit einem Inkrement von 1. Firebird wird:

1. den aktuellen Generatorwert holen;
2. ihn um 1 inkrementieren (und speichern);
3. den inkrementierten Wert zurückliefern

Client SQL

```
SELECT GEN_ID( <GeneratorName>, 1 ) FROM RDB$DATABASE;
```

Die neue, für Firebird 2 empfohlene Syntax, ist völlig verschieden:

```
SELECT NEXT VALUE FOR <SequenceName> FROM RDB$DATABASE;
```

PSQL

```
<intvar> = GEN_ID( <GeneratorName>, 1 );
```

Für Firebird 2 und höher bevorzugte Syntax:

```
<intvar> = NEXT VALUE FOR <SequenceName>;
```

Einen Generator direkt auf einen bestimmten Wert setzen („Update“)

Client SQL

```
SET GENERATOR <GeneratorName> TO <NewValue>;
```

Dies ist nützlich, um einen Generator auf einen anderen als den Standardwert 0, nach der Erzeugung vorzubelegen, z.B. in einem Skript, um die Datenbank zu erzeugen. Genau wie CREATE GENERATOR ist dies ein DDL- und kein DML-Befehl.

Für Firebird 2 und höher bevorzugte Syntax:

```
ALTER SEQUENCE <SequenceName> RESTART WITH <NewValue>;
```

PSQL

```
GEN_ID( <GeneratorName>, <NewValue> - GEN_ID( <GeneratorName>, 0 ) );
```

> **Warnung**
>
> Dies ist mehr ein mieser kleiner Trick um etwas zu tun, was man in SPs und Triggern niemals tun sollte: Generatoren (über-)schreiben. Sie sind zum Generieren (Lesen) und nicht zum Setzen (Schreiben) da.

Einen Generator löschen („Delete“)

Client SQL

```
DROP GENERATOR <GeneratorName>;
```

Für Firebird 2 und höher bevorzugte Syntax:

```
DROP SEQUENCE <SequenceName>;
```

PSQL

Nicht möglich, es sei denn... (Gleiche Erklärung wie bei CREATE: man kann - oder besser: *sollte* keine Änderung an den Metadaten in PSQL vornehmen).

Einen Generator zu Löschen gibt den von ihm belegten Platz, für die Verwendung durch einen neuen Generator, nicht wieder frei. In der Praxis stört dies kaum, da kaum eine Datenbank die Zigtausend Generatoren braucht, die Firebird zulässt, so dass es immer noch genügend Platz für neue gibt. Sollte die Datenbank aber doch Gefahr laufen, die 32767er Grenze zu erreichen, kann man den verbrauchten Platz durch einen Backup-Restore-Zyklus wiedergewinnen. Dies wird die RDB$GENERATORS-Tabelle komprimieren, unter Zuweisung einer neuen, lückenlosen Reihe von IDs. Abhängig von der Situation kann die wiederhergestellte Datenbank unter Umständen auch weniger Seiten zur Speicherung der Generatorwerte brauchen.

Generatoren in älteren IB- und Firebird-Versionen löschen

Sowohl InterBase 6 und frühere als auch frühe Prä-1.0-Firebird-Versionen kennen keinen DROP GENERATOR-Befehl. Die einzige Möglichkeit zum Löschen eines Generators in diesen Versionen ist:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = '<GeneratorName>';
```

...gefolgt von einem Backup und Restore.

In diesen Versionen war es durch die auf wenige hundert begrenzte Anzahl verfügbarer Generatoren wahrscheinlicher, dass man den Platz für gelöschte Generatoren durch Backup/Restore zurückgewinnen musste.

Generatoren zum Erzeugen eindeutiger Datensatz-IDs

Wozu überhaupt Datensatz-IDs?

Die Beantwortung dieser Frage würde den Rahmen diese Artikels deutlich sprengen. Derjenige, der keinen Sinn darin sieht, eine eindeutige Identifikationsmöglichkeit jedes Datensatzes in jeder Tabelle zu haben, oder dem das Konzept von "bedeutungslosen" oder "Surrogat"-Schlüsseln im allgemeinen missfällt, sollte das folgende Kapitel wohl besser überspringen...

Einer für alle oder einer für jede?

OK, du willst also Datensatz-IDs. { Anm.d.Autors: Glückwunsch! :-) }

Eine grundsätzliche, weitreichende Entscheidung muss gefällt werden: Benutzt man einen einzelnen Generator für alle Tabellen, oder jeweils einen Generator pro Tabelle. Dies ist dir überlassen - man sollte aber folgendes in Betracht ziehen:

Mit dem „Einer für alle“-Ansatz:

- + braucht man nur einen einzlnen Generator für alle IDs
- + hat man einen Integerwert, der den Datensatz nicht nur in seiner Tabelle, sondern in der gesamten Datenbank eindeutig identifiziert
- - hat man weniger verfügbare Generatorwerte pro Tabelle (das sollte mit 64bit-Generatoren nicht wirklich ein Problem sein)
- - bekommt man es bald mit unhandlich großen ID-Werten zu tun, selbst in z.B. kleinen Nachschlagetabellen mit nur einer Handvoll Einträgen
- - hat man höchstwahrscheinlich Lücken in den IDs einer Tabelle, da die ID-Werte über alle Tabellen verteilt werden

Mit dem „Einer für jede“-Ansatz:

- - muss man für jede ID-fähige Tabelle in der Datenbank einen eigenen Generator anlegen

- - braucht man immer die Kombination aus ID und Tabellenname zur eindeutigen Identifizierung des Satzes in der Datenbank
- + hat man einen einfachen und robusten „Einfügezähler“ pro Tabelle
- + hat man eine chronologische Sequenz pro Tabelle: Findet man eine Lücke in den IDs, stammt sie entweder von einem DELETE oder einem schiefgegangenen INSERT

Kann man Generatorwerte wiederverwenden?

Nun ja, technisch gesehen *kann* man das. Aber NEIN, man sollte es nicht. Niemals. NIE-NIE-NIEMALS. Nicht nur würde dies die schöne chronologische Reihenfolge der IDs zerstören (man kann das "Alter" eines Datansatzes nicht mehr an Hand der ID abschätzen), je mehr man darüber nachdenkt um so mehr Kopfschmerzen bereitet es. Abgesehen davon ist es ein völliger Widerspruch zum Konzept eindeutiger Datensatz-IDs.

Solange man also keine wirklich guten Gründe hat, Generatorwerte zu "recyclen", und einen wohlüberlegten Mechanismus besitzt, um dies in Mehrbenutzer/Multi-Transaktionsumgebungen sicher zu machen, FINGER WEG!

Generatoren für IDs oder Auto-Increment-Felder

Einem neu eingefügten Datensatz eine ID (im Sinne einer eindeutigen "Seriennummer") zu geben ist einfach zu bewerkstelligen unter Verwendung eines Generators und eines BEFORE INSERT-Triggers, wie wir im Folgenden sehen werden. Wir starten mit einer Tabelle TTEST mit einer Spalte ID, deklariert als Integer. Unser Generator heisst GIDTEST.

Before Insert Trigger, Version 1

```
CREATE TRIGGER trgTTEST_BI_V1 for TTEST
active before insert position 0
as
begin
  new.id = gen_id( gidTest, 1 );
end
```

Probleme mit Trigger Version 1:

Dieser erledigt die Arbeit - aber er "verschwendet" auch jedes mal einen Generatorwert, wenn im INSERT-Befehl bereits ein generierter Wert für die ID übergeben wurde. Es wäre also effektiver, nur dann einen neuen Wert zu generieren, wenn nicht bereits einer im INSERT-Befehl enthalten war:

Before Insert Trigger, Version 2

```
CREATE TRIGGER trgTTEST_BI_V2 for TTEST
active before insert position 0
as
begin
  if (ne    w.idisnullthen
  begin
    new.id = gen_id( gidTest, 1 );
```

```
  end
end
```

Probleme mit Trigger Version 2:

Manche Zugriffskomponenten haben die "dumme Angewohnheit", alle Spaltenwerte in einem Insert-Befehl vorzubelegen. Die Felder, die man nicht explizit setzt, bekommen Vorgabewerte - üblicherweise 0 für Integer-Spalten. In diesem Falle würde der obige Trigger nicht funktionieren: Er würde sehen, dass die ID-Spalte nicht den *Zustand* NULL, sondern den *Wert* 0 hat, und würde deshalb keine neue ID generieren. Man könnte den Satz dennoch speichern - aber nur einen... der zweite würde fehlschlagen. Es ist ohnehin eine gute Idee, die 0 als normalen ID-Wert zu "verbannen", allein schon um Verwechslungen zwischen NULL und 0 zu vermeiden. Man könnte z.B. einen speziellen Datensatz mit einer ID von 0 zur Speicherung der eigenen Vorgabewerte jeder Spalte in der Tabelle verwenden.

Before Insert Trigger, Version 3

```
CREATE TRIGGER trgTTEST_BI_V3 for TTEST
active before insert position 0
as
begin
  if ((new.id is null) or (new.id = 0)) then
  begin
    new.id = gen_id( gidTest, 1 );
  end
end
```

Nun, da wir einen robust funktionierenden ID-Trigger haben, werden die folgenden Absätze erläutern, warum man den meistens gar nicht braucht:

Das Grundproblem mit IDs, die in Before-Insert-Triggern zugewiesen werden, ist, dass sie die IDs serverseitig erzeugen, *nachdem* man den Insert-Befehl zum Server geschickt hat. Das heißt schlicht und ergreifend, dass es *keinen* sicheren Weg gibt, von der Client-Seite aus zu erfahren, welche ID für den gerade erzeugten Satz vergeben wurde.

Man könnte nach dem INSERT den aktuellen Stand des Generators abfragen, aber im Mehrbenutzerbetrieb kann man nicht wirklich sicher sein, dass es die ID des eigenen Datensatzes ist (wegen der Transaktionskontrolle).

Generiert man aber einen neuen Generatorwert *vorher*, und füllt die ID-Spalte im Insert-Befehl mit diesem Wert, dann kann man den Datensatz einfach mit einem "Select ... where ID=<GenWert>" aus der Datenbank holen, um z.B. zu sehen, welche Vorgabewert greifen oder welche Spalten durch Insert-Trigger verändert wurden. Dies funktioniert deshalb besonders gut, weil man üblicherweise einen eindeutigen Primärindex für die ID-Spalte hat, und das sind so ungefähr die schnellsten Indizes, die man kriegen kann - sie sind unschlagbar in punkto Selektivität, und meist auch kleiner als Indizes für Textfelder vom Typ CHAR(n) (gilt für n>8, abhängig von Zeichensatz und Sortierreihenfolge)

Fazit des Ganzen:

Man sollte immer einen Before Insert-Trigger erzeugen, um absolut sicher zu sein, dass jeder neue Datensatz eine eindeutige ID erhält, selbst wenn im Insert-Befehl keine übergeben wurde.

Hat man eine SQL-mäßig "geschlossene" Datenbank (d.h. die eigene Applikation ist die einzige Quelle neuer Datensätze), dann kann man den Trigger weglassen. Dann muss man aber IMMER einen Generatorwert vor dem Insert holen und ihn im Insert-Befehl mitübergeben. Das selbe gilt selbstverständlich für Inserts, die aus Stored Procedures oder Triggern heraus erfolgen.

Was man sonst noch mit Generatoren machen kann

Hier gibt es noch ein paar Anregungen für den Gebrauch von Generatoren für andere Zwecke als das Erzeugen von Datensatz-IDs.

Generatoren verwenden, um z.B. Transferdateien eindeutig zu kennzeichnen

Eine "klassische" Anwendung von Generatoren ist es, eindeutige, aufeinanderfolgende Werte zu erzeugen für - na ja, alles in der Applikation, abgesehen von den oben diskutierten Datensatz-IDs. Exportiert die Anwendung z.B. Daten zu anderen Systemen, kann man Generatorwerte sicher zur eindeutigen Bezeichnung jedes Transfers benutzen. Dies hilft enorm bei der Fehlersuche in solchen Datenschnittstellen (und anders als die meisten der folgenden Anwendungen funktioniert es sicher und robust).

Generatoren als "Benutzungszähler" für StoredProcs als Statistikgrundlage

Stelle dir vor, du hast gerade ein fantastisches neues Feature mittels einer Stored Procedure erstellt. Jetzt spielst du die neue Version beim Kunden ein und möchtest später wissen, ob und wie oft die Kunden dieses Feature wirklich benutzen. Simpel: man nehme einen speziellen Generator, der nur in dieser StoredProc hochgezählt wird, und das war`s... mit der kleinen Einschränkung, dass man nicht wissen kann, wie viele Transaktionen mit Aufrufen der SP durch ein Rollback nicht zu Ende gebracht wurden. In jedem Falle aber weiss man dann, wie oft Benutzer *versucht* haben, die SP zu benutzen. :-)

Man könnte diese Methode noch verfeinern, in dem man zwei Generatoren benutzt: einer wird direkt am Start der Prozedur erhöht, der zweite ganz am Ende vor dem EXIT. Haben beide nach einer Zeit den selben Wert, dann ist innerhalb der SP nie etwas schiefgegangen etc. Natürlich weiss man immer noch nicht, wieviele SP-Aufrufe einem Rollback der aufrufenden Transaktion zum Opfer gefallen sind.

Generatoren zur Simulation von „Select count(*) from...“

Es ist ein bekanntes Problem von InterBase und Firebird, dass ein SELECT COUNT(*) (ohne Where-Klausel) bei einer wirklich grossen Tabelle eine ganze Weile zur Durchführung benötigt, da der Server "zu Fuss" durchzählen muss, wie viele Sätze sich zum Zeitpunkt des Aufrufs gerade in der Tabelle befinden (Stichwort: Multigenerationsarchitektur). *Theoretisch* liesse sich dieses Problem einfach durch den Einsatz von Generatoren umgehen:

- man nehme einen speziellen "Satzzähler"-Generator;
- man erzeuge einen Before Insert Trigger, der ihn erhöht
- und einen After Delete Trigger, der ihn wieder runterzählt.

Das funktioniert wunderbar und macht ein "volles" Durchzählen der Datensätze überflüssig - man fragt einfach den aktuellen Generatorwert ab. Die Betonung liegt hier auf *theoretisch*, denn das ganze geht den Bach runter,

sobald Insert-Befehle schiefgehen, denn wie gesagt liegen Generatoren *ausserhalb jeder Transaktionskontrolle*. Insert-Befehle können durch Constraints (eindeutige Index-Verletzungen, NOT NULL-Felder enthalten NULL etc.) oder durch andere Metadaten-Einschränkungen schiefgehen, oder einfach weil die aufrufende Transaktion mit einem Rollback endet. Man hat keine Datensätze in der Tabelle und trotzdem steigt der Zähler-Generator.

Es kommt also drauf an - wenn man den ungefähren Prozentsatz schieflaufender Inserts kennt (man kann dafür ein "Gefühl" entwickeln), und es nur um eine grobe *Abschätzung* der Anzahl der Datensätze geht, dann kann diese Methode hilfreich sein, obwohl sie nicht exakt ist. Von Zeit zu Zeit kann man ein "normales" Durchzählen der Sätze durchführen, um den Generator wieder auf den richtigen Wert zu setzen ("Re-Synchronisation" des Generators), so dass man den Fehler in Grenzen halten kann.

Es gibt Situationen, wo Kunden glücklich leben können mit einer Aussage wie "es gibt *ungefähr* 2,3 Millionen Datensätze in der Tabelle", die sie sofort auf einen Mausklick hin erhalten, einen aber erschiessen würden, wenn sie 10 Minuten oder mehr warten müssen, um zu erfahren, dass es exakt 2.313.498.229 Datensätze sind...

Generatoren zum Überwachen und/oder Steuern lange laufender Stored Procedures

Hat man Stored Procedures, die z.B. Auswertungen auf grossen Tabellen oder über komplexe Joins fahren, dann können diese ganz schön lange brauchen. Hier können Generatoren auf zweierlei Weise helfen: Sie können einen Fortschrittszähler liefern, den man zyklisch vom Client aus abfragen kann, und sie können benutzt werden, um die Ausführung abzubrechen:

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;

set term ^;

CREATE PROCEDURE spTest (...)
AS
BEGIN
  (...)
  for select <viele Daten die lange zur Auswertung brauchen>
  do begin
    GEN_ID(gen_spTestProgress,1);

    IF (GEN_ID(gen_spTestStop,0)>0) THEN Exit;

    (...hier die normale Abarbeitung...)
  end
END^
```

Nur ein grober Entwurf, aber das Konzept sollte erkennbar sein. Von der Client-Seite aus kann man ein GEN_ID(gen_spTestProgress,0) asynchron zur Ausführung der SP aufrufen (z.B. in einem zweiten Thread), um zu sehen, wie viele Sätze bereits abgearbeitet sind, und diesen Wert in einem Fortschrittsbalken anzeigen. Und man kann mittels GEN_ID(gen_spTestStop,1) die SP jederzeit von "aussen" abbrechen.

Auch wenn dies sehr hilfreich sein kann, hat es eine starke Einschränkung: *Es ist nicht sicher im Mehrbenutzer-Betrieb*. Würde die SP parallel von zwei Transaktionen aus aufgerufen, würde der Fortschrittszähler gestört - beide Aufrufe würden den gleichen Generator erhöhen, und das Resultat wäre unbrauchbar. Schlimmer noch, eine Inkrementierung des STOP-Generators würde die SP in *beiden* Transaktionen beenden. Aber für z.B. Monatsauswertungen, die von einem einzigen Modul im Batch-Betrieb gefahren werden, kann dies akzeptabel sein - wie üblich hängt es von den Randbedingungen ab.

Will man diese Technik einsetzen, um vom Benutzer jederzeit aufrufbare SPs zu steuern, muss durch andere Mechanismen gewährleistet werden, dass die SP nicht zeitgleich mehrmals ausgeführt werden kann. Darüber sinnierend kam mir die Idee, dafür einen weiteren Generator einzusetzen: nennen wir ihn gen_spTestLocked (unter Annahme des Startwerts von 0 natürlich):

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;
CREATE GENERATOR gen_spTestLocked;

set term ^;

CREATE PROCEDURE spTest (...)
AS
DECLARE VARIABLE lockcount INTEGER;
BEGIN
  lockcount = GEN_ID(gen_spTestLocked,1);
    /* allererster Schritt: Erhöhen des Blockade-Generators */

  if (lockcount=1) then /* _wir_ haben die Sperre, weitermachen */
  begin
    (..      .hierdernormaleProzedurrumpf...
  end

  lockcount = GEN_ID(gen_spTestLocked,-1); /* Erhöhung rückgängig machen */

  /* sicherstellen dass der Generator auch bei Ausnahmen (Exceptions) im
     Prozedurrumpf jederzeit sauber zurückgesetz wird: */

  WHEN ANY DO
    lockcount = GEN_ID(spTestLocked,-1); /* s.o. */
  exit;
END^
```

Hinweis: Ich bin mir nicht 100%ig sicher, ob dies im Mehrbenutzerbetrieb jederzeit sauber funktioniert, aber es sieht recht "schusssicher" aus - so lange kein EXIT im normalen Prozedurrumpf vorkommt, denn dann würde die Prozedur mittendrin verlassen werden und der Blockade-Generator würde inkrementiert stehenbleiben. Die WHEN ANY-Klausel behandelt Ausnahmen, aber keine normalen EXITs. Dann müsste man den Generator von Hand zurücksetzen - aber man könnte ihn auch im Prozedurrumpf direkt vor dem EXIT herunterzählen. Mit den geeigneten Sicherheitsvorkehrungen fällt mir keine Situation ein, wo dieser Mechanismus fehlschlagen würde - falls dir eine einfällt, lass es uns wissen!

Anhang A: Dokumentenhistorie

Die genaue Dokumentenhistorie wird fortgeschrieben im `manual` -Modul in unserem CVS-Baum; siehe http://sourceforge.net/cvs/?group_id=9028

Versionsgeschichte

0.1	4 Apr 2006	FI	Erste Ausgabe.
0.2	7 May 2006	PV	SEQUENCE-Syntax und andere Firebird 2-Infos hinzugefügt. Informationen hinzugefügt über: Die Wichtigkeit des Client-Dialekts, den SHOW GENERATOR Befehl und Verwandte, das Löschen von Generatoren und die Wiedergewinnung des dadurch verlorenen Speichers. Folgende Kapitel mehr oder weniger heftig bearbeitet und erweitert: *Wo werden Generatoren gespeichert?*, *Was ist der maximale Wert eines Generators?*, *Wie viele Generatoren...?*, *Verwendung der Generatorbefehle*. Weitere Bearbeitung, Ergänzungen und Korrekturen in verschiedenen Kapiteln, größtenteils in der ersten Hälfte des Dokuments. Leichte Überarbeitung in der zweiten Hälfte (beginnend mit *Generatoren zur Erzeugnug eineindeutiger Datensatz-IDs*).
0.2-de	4. Dezember 2006	FI	Deutsche Übersetzung basierend auf der englischen Dokumentenversion 0.2.

Anhang B: Lizenzhinweis

Die Original-Dokumentation trägt den Titel *Firebird Generator Guide*.

Der ursprünglich Autor der Original-Dokumentation ist: Frank Ingermann.

Co-Autor: Paul Vinkenoog – siehe document history.