Использование генераторов в СУБД Firebird

Как и когда нужно использовать генераторы в СУБД Firebird

Frank Ingermann

29 октября 2006 – Версия документа 0.2-ru

Перевод документа на русский язык: Сергей Ковалёв

Содержание

Введение

О чем эта статья?

Эта статья объясняет, что такое генераторы в СУБД Firebird, как и почему вы должны использовать их. В этой статье предпринята попытка собрать всю информацию, относящуюся к генераторам, в одном месте.

Кому нужно это прочитать?

Прочтите эту статью, если вы:

- не знакомы с концепцией генераторов;
- имеете вопросы по использованию генераторов;
- хотите создать поле типа Integer, наподобие поля с «автоинкрементом», которые есть в других СУБД;
- ищите примеры того, как использовать генераторы для идентификаторов (ID) или для других задач;
- хотите узнать, что в СУБД Firebird является аналогом понятия «sequence» из СУБД Oracle.

Основные положения о генераторах

Что такое генератор?

Думайте о генераторе, как о «потокобезопасном» («thread-safe») целочисленном счетчике, который расположен внутри базы данных Firebird. Вы можете создать его, задав имя:

```
CREATE GENERATOR GenTest;
```

Затем вы можете получать его текущее значение, увеличивать его или уменьшать точно так же, как и переменную «var i: integer» в Delphi, однако не всегда можно просто установить определенное значение, а затем получить то же самое значение, как вы ожидаете, - генератор находится внутри базы данных, но *вне механизма управления транзакциями*.

Что такое последовательность (sequence)?

«Последовательность» («sequence») - это официальный термин SQL для обозначения того, что в СУБД Firebird называется генератором. Поскольку СУБД Firebird постоянно стремиться к большему соответствию стандарту SQL, то в СУБД Firebird 2 и более поздних версиях ключевое слово SEQUENCE может быть использован как синоним GENERATOR. Фактически, рекомендуется использовать синтаксис SEQUENCE во вновь создаваемом коде.

Хотя слово «последовательность» подразумевает серию генерируемых значений, в то время как «генератор» подразумевает прямую ссылку на фабрику по производству значений, в СУБД Firebird *нет никаких различий* между генератором и последовательностью. Просто это два названия для одного и того же объекта базы данных. Вы можете создавать генератор и получать доступ к нему с помощью синтаксиса последовательности, и наоборот.

Вот предпочтительный синтаксис для создания генератора/последовательности в СУБД Firebird 2:

```
CREATE SEQUENCE SeqTest;
```

Где хранятся генераторы?

Декларации генераторов хранятся в системной таблице RDB$GENERATORS. Однако, их *значения* хранятся на специальных зарезервированных страницах внутри базы данных. Вы никогда не получите доступ к этим значениям напрямую. Вы можете получить к ним доступ с помощью специальных встроенных функций, которые будут обсуждаться далее в этом руководстве.

> **Внимание**
>
> Приведенная в этом разделе информация предназначена только для общего ознакомления. Общее правило таково: вы не должны обращаться к системным таблицам напрямую. Не пытайтесь создавать или изменять генераторы путем изменения таблицы RDB$GENERATORS. (Хотя оператор SELECT и не наделает бед.)

Структура системной таблицы RDB$GENERATORS следующая:

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

И для СУБД Firebird 2.0 и более старших версий:

- RDB$DESCRIPTION BLOB subtype TEXT

Обратите внимание, что GENERATOR_ID является идентификатором для каждого генератора (о чем и говорит это имя), а *не* его значением. Также, не позволяйте вашему приложения хранить идентификатор для будущего использования в качестве маркера (handle) генератора. Это не имеет смысла (так как *имя* и является маркером), идентификатор может измениться после цикла резервного копирования/восстановления базы данных. Флаг SYSTEM_FLAG равен 1 для генераторов, используемых внутри СУБД, и `NULL` или 0 для всех, созданных вами.

Теперь давайте взглянем на таблицу RDB$GENERATORS, когда определен единственный генератор:

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

Примечание для СУБД Firebird 2

- В СУБД Firebird 2 введен дополнительный системный генератор, называемый RDB $BACKUP_HISTORY. Он используется для новой утилиты NBackup.
- Несмотря на то, что синтаксис SEQUENCE предпочтителен, системная таблица RDB$GENERATORS и ее поля не были переименованы в СУБД Firebird 2.

Каково максимальное значение генератора?

Генераторы хранят и возвращают 64-битные значения во всех версиях СУБД Firebird. Это дает диапазон значений:

-2^{63} .. $2^{63}-1$ или -9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807

Таким образом, если вы используете генератор с начальным значением 0 для заполнения поля типа NUMERIC(18) или BIGINT (оба типа представлены 64-битным целым), и вы хотели бы добавлять 1000 строк в секунду, то пройдет около 300 миллионов лет (!), прежде чем значения выйдут за указанный диапазон. Поскольку довольно маловероятно, что человечество еще будет существовать на планете в это время (и оно все еще будет использовать базы данных СУБД Firebird), то не следует об этом беспокоиться.

Но хочу предупредить. СУБД Firebird понимает два «диалекта» SQL: диалект 1 и диалект 3. Новые базы данных должны всегда создаваться с диалектом 3, который является более мощным в ряде аспектов. Диалект 1 является диалектом совместимости, он используется только для полученных в наследство баз данных, которые были созданы в InterBase 5.6 или в более ранних версиях.

Одним из различий этих двух диалектов является то, что диалект 1 не имеет поддержки встроенных 64-битных целочисленных типов. Поля типа NUMERIC(18), например, хранятся во внутреннем представлении как DOUBLE PRECISION, который является типом с плавающей точкой. Наибольший целочисленный тип в диалекте 1 - это 32-битный INTEGER.

В диалекте 1, как и в диалекте 3, генераторы являются 64-битными. Но если вы присвоите сгенерированное значение полю типа INTEGER в базе данных диалекта 1, оно будет урезано до младших 32 бит, в результате давая диапазон:

-2^{31} .. $2^{31}-1$ или -2.147.483.648 .. 2.147.483.647

Хотя сам генератор может давать значения и 2.147.483.647, и 2.147.483.648, и так далее, урезанное значение вызовет повторение значений в этом месте, давая *эффект* 32-битного генератора.

В описанной выше ситуации, при 1000 вставках в секунду, заполняемое генератором поле переберет все значения через 25 *дней* (!!!), и за этим действительно нужно следить. 2^{31} - это очень много, но насколько это много, зависит от ситуации.

Замечание

В диалекте 3, если вы присваиваете значение генератора полю типа INTEGER, то все идет хорошо, если значение умещается в 32-битный диапазон. Но, как только этот диапазон будет превышен, вы получите ошибку переполнения: диалект 3 более строг в проверке диапазона по сравнению с диалектом 1!

Диалект клиентов и значения генераторов

При общении с сервером СУБД Firebird клиент может установить как диалект 1, так и диалект 3, независимо от того, к какой базе данных он подключен. Именно диалект клиента, а *не* диалект базы данных, определяет как СУБД Firebird передает значение генератора клиенту:

- Если диалект клиента 1, сервер возвращает клиенту значение генератора в виде урезанного 32-битного целого. Но внутри базы данных сгенерированные значения остаются 64-битными, и они не зацикливаются после достижения 2^{31}-1 (даже если это так выглядит на стороне клиента). Это верно для баз данных и диалекта 1, и диалекта 3.
- Если диалект клиента 3, сервер передает полное 64-битное значение клиенту. Опять-таки, это верно для баз данных и диалекта 1, и диалекта 3.

Сколько генераторов доступно для одной базы данных?

Начиная с СУБД Firebird версии 1.0, количество генераторов, которые вы можете создать в одной базе данных, ограничено только максимальным значением идентификатора (ID) в системной таблице RDB $GENERATORS. Так как это SMALLINT, то его максимальное значение равно 2^{15}-1 или 32767. Первое значение идентификатора всегда 1, так что полное количество генераторов не может превышать 32767. Как указывалось выше, в базе данных есть восемь или девять системных генераторов, таким образом остается, по крайней мере, 32758 для ваших собственных генераторов. Это должно быть достаточно для любого практического приложения. И так как количество декларируемых вами генераторов, не влияет на производительность, то вы можете чувствовать себя свободным и использовать столько генераторов, сколько хотите.

Старые версии InterBase и Firebird

В ранних версия СУБД Firebird до версии 1.0, также, как и в СУБД InterBase, для хранения значений генераторов использовалась только одна страница базы данных. Поэтому, количество доступных генераторов было ограничено размером страницы базы данных. Следующая таблица перечисляет, сколько генераторов (включая системные) вы можете использовать в этих версиях СУБД InterBase и Firebird (спасибо Полу Ривзу [Paul Reeves] за предоставление оригинальной информации):

Версия	**Размер страницы**			
	1K	*2K*	*4K*	*8K*
InterBase < v.6	247	503	1015	2039
IB 6 и версии Firebird до 1.0	123	251	507	1019
Все последующие версии Firebird	32767			

В СУБД InterBase до версии 6 генераторы были только 32-битные. Это объясняет, почему в старых версиях можно хранить ровно в два раза больше генераторов при одном и том же размере страницы.

Внимание

СУБД InterBase, по крайней мере, до версии 6.01 включительно, успешно позволяет вам «создавать» генераторы до тех пор, пока их общее количество не достигнет 32767. Что произойдет, если вы обратитесь к генератору с идентификатором б*о*льшим, чем номер, указанный в приведенной выше таблице, зависит от версии:

- InterBase 6 генерирует ошибку «invalid block type» (неверный тип блока), поскольку вычисляемое значение расположено вне страницы, которая зарезервирована под генераторы.
- В более ранних версиях, если вычисляемое место расположено вне страниц базы данных, возвращается ошибка. В противном случае, если генератор только *читается* (без инкремента), его значение представляет из себя то, что расположено по вычисленному месту (вычисленной странице базы данных). Если значение было *записано*, при этом перезапишутся данные, расположенные в вычисленном месте. Иногда это приводит к немедленной ошибке, но чаще всего это приводит к молчаливой порче вашей базы данных.

Генераторы и транзакции

Как уже было сказано, генераторы находятся вне механизма управления транзакциями. Это означает, что вы не можете безопасно «откатить» генератор внутри транзакции. Одновременно с вашей транзакцией может существовать другая транзакция, выполняющая в то же самое время изменение значения генератора. Так что, если вы запросили значение генератора, думайте о нем, как об «ушедшем навсегда».

Когда вы запускаете транзакцию, а затем вызываете генератор и получаете значение (скажем, 5), генератор останется на том же самом значении, **даже если вы выполните откат транзакции (!)**. Даже *не смейте думать* так: «Ну, хорошо, когда я выполню откат, я просто вместе с этим выполню GEN_ID(mygen,-1), чтобы снова установить генератор в значение 4». Чаще всего это может сработать, но это *не безопасно*, поскольку другая конкурентная транзакция может изменить значение генератора в период времи между вашим обращением к генератору и откатом транзакции. По этой же причине не имеет смысла получать текущее значение с помощью GEN_ID(mygen,0), а затем увеличивать это значение на стороне клиента.

Операторы SQL для генераторов

Обзор операторов

Имя генератора должно являться обычным идентификатором метаданных базы данных: максимум 31 символ, без специальных символов за исключением символа подчеркивания «_» (если вы не используете регистрозависимые идентификаторы в кавычках). Команды и операторы SQL, применяемые к генераторам, перечислены ниже. Их использование более подробно будет описано в разделе *Использование операторов для генераторов*.

Операторы DDL (Data Definition Language - язык определения данных):

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

Операторы DML (Data Manipulation Language - язык манипуляции данными) в клиентском SQL:

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

Операторы DML в PSQL (Procedural SQL - процедурный SQL - расширение языка, используемое в хранимых процедурах и триггерах):

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

Синтаксис, рекомендованный для Firebird 2

Хотя в СУБД Firebird 2 все еще полностью поддерживается традиционный синтаксис, существует рекомендуемый DDL-эквивалент для СУБД Firebird 2:

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

А для операторов DML:

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

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

В настоящее время рекомендуемый синтаксис не поддерживает шаг изменения (инкремент), отличный от 1. Это ограничение будет снято в будущих версиях. Пока используйте GEN_ID, если вы хотите использовать другое значение для шага изменения значения.

Использование операторов для генераторов

Доступность операторов и функций зависит от того, где вы их используете:

- Клиентский SQL – используемый вами язык, когда вы, в качестве клиента, общаетесь с сервером СУБД Firebird.
- PSQL – язык программирования на стороне сервера, используемый в хранимых процедурах и триггерах СУБД Firebird.

Создание генератора («Insert»)

Клиентский SQL

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

Предпочтительно для СУБД Firebird 2 и старше:

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

PSQL

Невозможно. Так как вы не можете изменять метаданные базы данных в хранимых процедурах (stored procedures) или триггерах, вы не можете создавать генераторы.

> **Замечание**
>
> В СУБД Firebird 1.5 и старше вы можете обходить это ограничение с помощью оператора EXECUTE STATEMENT.

Получение текущего значения («Select»)

Клиентский SQL

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

Этот синтаксис является единственной возможностью получить текущее значение генератора в СУБД Firebird 2.

> **Замечание**
>
> В утилите для СУБД Firebird *isql* есть две дополнительные команды для получения текущего значения генератора:
>
> ```
> SHOW GENERATOR <GeneratorName>;
> SHOW GENERATORS;
> ```
>
> Первая показывает текущее значение конкретного генератора. Вторая команда делает то же самое для всех не системных генераторов базы данных.
>
> Предпочтительный для СУБД Firebird 2 эквивалент, как вы можете предположить:
>
> ```
> SHOW SEQUENCE <SequenceName>;
> SHOW SEQUENCES;
> ```
>
> Пожалуйста, обратите внимание, что команды SHOW... доступны только в инструменте isql. В отличие от GEN_ID, вы не можете использовать их из других клиентов (если эти клиенты не используют isql в качестве клиента).

PSQL

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

СУБД Firebird 2: тот же самый синтаксис.

Генерация следующего значения («Update» + «Select»)

Так же, как и в случае получения текущего значения, это выполняется с помощью GEN_ID, но в этом случае вы используете значение шага, равное 1. При этом СУБД Firebird:

1. получает текущее значение генератора;
2. увеличивает его на 1;
3. возвращает измененное значение.

Клиентский SQL

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

Новый синтаксис, который предпочтителен для СУБД Firebird 2, полностью отличается:

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

PSQL

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

Предпочтительно для СУБД Firebird 2 и старше:

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

Прямое указание определенного значения генератора («Update»)

Клиентский SQL

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

Это удобно для установки в генераторе значения, отличного от 0 (которое является значением по умолчанию после его создания), например, в скрипте для создания базы данных. Аналогично CREATE GENERATOR, это оператор DDL (не DML).

Предпочтительный синтаксис для СУБД Firebird 2 и старше:

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

PSQL

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

> **Внимание**
>
> Это больше похоже на «грязный трюк» в попытке сделать то, что в обычных условиях сделать нельзя и не должно в хранимых процедурах и триггерах: *установка* значений генераторов. В них можно *получать*, а не *устанавливать* значения.

Удаление генератора («Delete»)

Клиентский SQL

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

Предпочтительно для СУБД Firebird 2 и старше:

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

PSQL

Невозможно, поскольку... (То же самое объяснение, что и для создания: вы не можете [или, скорее, не должны] изменять метаданные в PSQL.)

Удаление генератора не освобождает место, которое он занимает, для использования нового генератора. На практике это редко приводит к проблемам, поскольку большинство баз данных не имеют десятков тысяч генераторов, которые позволяет создавать СУБД Firebird. Но если ваша база данных *рискует* пре-

высить 32767 генераторов, вы можете освободить место неиспользуемых генераторов путем выполнения операции резервного копирования/восстановления базы данных (backup/restore). При этом аккуратно упаковывается таблица RDB$GENERATORS, переназначаются идентификаторы (ID), образуя непрерывную последовательность. В зависимости от ситуации, восстановленной базе данных может понадобиться меньше страниц для хранения значений генераторов.

Удаление генераторов в старых версиях IB и Firebird

СУБД InterBase 6 и более ранние версии, точно так же, как и ранние версии СУБД Firebird, предшествующие версии 1.0, не имеют команды DROP GENERATOR. Единственным способом удалить генератор является оператор:

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

...с последующим циклом резервирования/восстановления базы данных (backup/restore).

Для этих версий СУБД с максимальным количеством генераторов около пары сотен гораздо более вероятно появление необходимости использовать место от удаленных генераторов.

Использование генераторов для создания уникальных идентификаторов строк

Зачем вообще нужны идентификаторы (ID)?

Ответ на этот вопрос выходит далеко за пределы темы этой статьи. Если вы не видите необходимости иметь общий, уникальный «маркер» (handle) для каждой строки внутри таблицы, или вам вообще не нравится идея «бессмысленных» или «суррогатных» (искусственных) ключей, вам, вероятно, лучше пропустить этот раздел...

Один для всего или один для каждого?

Хорошо... Итак, вы хотите использовать идентификаторы. { примечание автора: мои поздравления! :-) }

Основной, наиболее важный выбор, который вам нужно сделать, - это использовать единый генератор для всех таблиц или один генератор для каждой таблицы. Это только ваш выбор, но примите во внимание следующие размышления.

В подходе «один для всех» вы:

- + нуждаетесь только в одном генераторе для всех ваших ID;
- + имеете одно целочисленное значение, которое не только однозначно идентифицирует строку внутри самой *таблицы*, но и внутри *всей базы данных*;
- - имеете меньшее количество значений ID на таблицу (на самом деле это не проблема для 64-битных генераторов...);

- - будете иметь дело с большими значениями ID, даже если, например, просматриваете таблицу с горсткой записей;
- - вероятно, будете видеть пробелы в последовательности ID каждой таблицы, так как значения генератора распределены между всеми таблицами.

В подходе «один для каждой» вы:

- - должны создавать генератор для каждого идентификатора таблицы вашей базы данных;
- - всегда должны комбинировать ID и имя таблицы для уникальной идентификации строки в таблице;
- + имеете простой и надежный «счетчик вставок» для таблицы;
- + имеете хронологическую последовательность для таблицы: если вы найдете пробелы в последовательности ID таблицы, то они появилась либо из-за удаления (DELETE), либо при неудачной вставке (INSERT).

Можно ли использовать значения генератора повторно?

В действительности – да, технически это *возможно*. В реальности – НЕТ, вы не должны. Никогда. Ни при каких условиях. Не только потому, что это может нарушить красоту хронологической последовательности (вы не сможете более судить о «возрасте» строки, взглянув на ее ID), а еще и потому, что повышается риск появления сильной головной боли, которую вы при этом получите. Более того, это абсолютно противоречит всей концепции уникального идентификатора строки.

Так что если у вас нет веских причин для повторного использования значений генератора и хорошо продуманного механизма, который безопасно сделает эту работу в многопользовательской/многотранзакционной среде, НЕ ДЕЛАЙТЕ ЭТОГО!

Генераторы для идентификации, или автоинкрементные поля

Указание ID для вновь вставляемой записи (в терминах уникальности - «серийный номер») легче всего выполнить с помощью генератора и триггера «перед вставкой» (Before Insert trigger), как мы увидим в этом разделе. Предположим, что у нас есть таблица TTEST с колонкой ID, объявленной как Integer. Имя нашего генератора: GIDTEST.

Триггер Before Insert, версия 1

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

Проблемы этого триггера (версии 1):

Он правильно делает свою работу, но он также «тратит» значение генератора в том случае, когда ID уже указан в операторе INSERT. Так что более эффективно было бы присваивать значение, когда ничего не было указано в INSERT:

Триггер Before Insert, версия 2

```
CREATE TRIGGER trgTTEST_BI_V2 for TTEST
active before insert position 0
as
begin
  if (new.id is null) then
  begin
    new.id = gen_id(gidTest, 1);
  end
end
```

Проблемы этого триггера (версии 2):

Некоторые компоненты доступа имеют «плохую привычку» автоматического заполнения колонок при выполнении INSERT. Если вы явно не указали, они устанавливают значение по умолчанию - обычно это 0 для колонок целочисленного типа. В этом случае приведенный выше триггер не будет работать: он будет обнаруживать, что колонка ID не находится в *состоянии* `NULL`, а имеет *значение* 0, и поэтому не будет генерировать новое значение ID. Вы сможете вставить запись, но только одну: вставка второй записи окончится неудачей. В любом случае, хорошей идеей будет запрет значения 0 в качестве обычного значения ID, чтобы предотвратить любую неоднозначность между `NULL` и 0. Вы можете использовать, например, специальную строку с ID, равным 0, для хранения записи «по умолчанию» в каждой таблице.

Триггер Before Insert, версия 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
```

Хотя теперь у нас есть понятный, работающий триггер для ID, следующие абзацы объяснят вам, почему вы чаще всего не захотите им пользоваться.

Основная проблема с присваиванием ID в триггере «перед вставкой» состоит в том, что значение генерируется на стороне сервера, *после того*, как вы отправили оператор добавления с клиента. Очевидно, это означает, что нет безопасного способа для клиента узнать, какой ID был сгенерирован для только что вставленной вами строки.

Вы можете попытаться получить значение генератора на стороне клиента после вставки, но в многопользовательской среде вы не можете быть уверены, что вы получили свой собственный ID строки (из-за вопросов с транзакциями).

Но если вы *сначала* получите новое значение генератора и передадите его при вставке, вы можете просто получить эту новую запись с помощью «Select ... where ID = <genvalue>», чтобы увидеть, какие умолчания были применены, или какие колонки были изменены триггерами при вставке. Это особенно хорошо работает, поскольку у вас обычно есть уникальный индекс для первичного ключа (Primary Key) по полю ID, а это наиболее быстрый индекс, который только может быть, - у него идеальная селективность

и он гораздо меньше, чем индекс для колонки CHAR(n) (для n>8, в зависимости от кодировки и порядка сортировки [collation]).

Из этого получаем вывод:

Вы должны создавать триггер «перед вставкой», чтобы быть абсолютно уверенным, что каждая строка получит уникальный ID, даже если значение ID не указано на стороне клиента в операторе вставки.

Если ваша база данных «закрыта для SQL» (то есть только ваше собственное приложение может стать источником вставки новых записей), то вы можете отказаться от триггеров, но при этом вы должны *всегда* получать новое значение генератора из базы данных прежде, чем выполните оператор вставки, и включать полученное значение в этот оператор. То же самое, конечно же, относится для вставок из триггеров и хранимых процедур.

Что еще делают с помощью генераторов

Здесь вы можете найти некоторые идеи, как использовать генераторы не только для генерации уникальных идентификаторов для строк.

Использование генераторов для получения, например, уникального номера передаваемого файла

«Классическое» использование генераторов должно гарантировать уникальность, последовательность чисел, скажем, для любого аспекта вашего приложения, отличного от идентификаторов строк, обсужденного ранее. Когда у вас есть приложение, которое передает данные в некоторую другую систему, вы можете использовать генераторы для безопасной идентификации одной сессии передачи, помечая ее сгенерированным значением. Это отлично помогает отследить проблемы в интерфейсе между двумя системами (и, в отличие от далее упомянутого, это работает безопасно и четко).

Генераторы, как «счетчик использований» для SP, обеспечивающие основную статистику

Представьте, что вы разработали в базе данных фантастическую новую возможность, которая реализована с помощью хранимой процедуры. Теперь вы обновили систему покупателя, и некоторое время спустя вы захотите узнать, *пользуются ли* клиенты этой возможностью и как часто. Это просто: создайте специальный генератор, который инкрементируется в этой хранимой процедуре и вы узнаете это (с учетом того ограничения, что вы не сможете узнать количество транзакций, которые были откачены после выполнения этой хранимой процедуры). Так что в этом случае вы, как минимум, узнаете как часто пользователи *пробуют* пользоваться вашей хранимой процедурой. :-)

В будущем вы сможете усовершенствовать этот метод и использовать два генератора: первый изменяется при каждом запуске хранимой процедуры, а второй изменяется при окончании работы процедуры, сразу перед выходом из нее (EXIT). Таким образом вы сможете посчитать, сколько попыток использования процедуры завершилось успехом: если оба генератора имеют одно и то же значение, то ни один вызов

хранимой процедуры не был неудачен. Конечно же, вы все еще не можете узнать, сколько раз транзакция (транзакции), в которой использовалась ваша процедура, была реально подтверждена (commit).

Генераторы, эмулирующие «Select count(*) from...»

В СУБД InterBase и Firebird существует известная проблема, когда SELECT COUNT(*) (без выражения Where) для очень большой таблицы может очень долго выполняться, так как сервер должен посчитать «вручную», сколько строк есть в таблице во время получения запроса. Теоретически, вы легко можете решить эту проблему с помощью генераторов:

- создайте специальный генератор для «подсчета строк»;
- создайте триггер «перед вставкой» (Before Insert), который увеличивает его;
- создайте триггер «после удаления» (After Delete), который уменьшает его.

Это прекрасно работает и отменяет необходимость в подсчете «полного» количества записей: мы просто получаем текущее значение генератора. Я специально подчеркну слово «*теоретически*», поскольку в целом картина портится, когда какой-нибудь оператор Insert будет отменен, а генераторы не *подчиняются механизму управления транзакциями*. Вставка может провалиться из-за ограничений (по уникальному ключу, поля NOT NULL, указаного как `NULL`, и т. п.) или по другим ограничениями метаданных, или просто потому, что транзакция, выполнявшая вставку, будет откачена. У вас не будет строк в таблице, а ваш счетчик вставок увеличится.

Так что, смотрите сами – когда вы хотите знать примерный процент вставок, завершившихся неудачей (вы можете частично «оценить» это), или вас интересует *примерное* количество записей, то этот метод может быть очень полезным, даже несмотря на то, что он и не точен. Время от времени вы можете выполнять «нормальный» подсчет записей и задавать генератору точное значение («синхронизировать» генератор), так что величина ошибки будет достаточно мала.

Существуют ситуации, когда клиенты достаточно счастливо живут с информацией, наподобие «существует *примерно* 2,3 миллиарда записей», появляющейся немедленно по щелчку мыши, но они готовы застрелить вас, если они ждут 10 минут или больше для того, чтобы увидеть, что существует ровно 2,313,498,229 строк...

Генераторы для мониторинга и/или управления долго работающей SP

Когда у вас есть хранимые процедуры, которые, например, генерируют отчет по большим таблицам и/или с помощью сложных соединений (join), они могут довольно долго выполняться. Генераторы могут быть полезны двумя способами: они могут предоставить вам «счетчик прогресса», который вы можете периодически опрашивать на стороне клиента и следить за работой процедуры, и они могут быть использованы для остановки хранимой процедуры:

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;

set term ^;

CREATE PROCEDURE spTest (...)
AS
```

```
BEGIN
  (...)
  for select <огромное количество данных с огромным временем обработки>
  do begin
    GEN_ID(gen_spTestProgress, 1);

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

    (...нормальная обработка данных...)
  end
END^
```

Это просто набросок, но вы должны уловить идею. С клиента вы можете выполнять GEN_ID(gen_spTestProgress, 0) асинхронно с реальной обработкой записей (например, в другом потоке [thread]), чтобы посмотреть, сколько записей было обработано, и отображать полученное значение в некотором окне прогресса. Так же вы можете выполнить GEN_ID(gen_spTestStop, 1), чтобы прекратить работу процедуры в любое время «извне» процедуры.

Хотя это может быть очень удобно, но здесь есть строгие ограничения: *это не безопасно в многопользовательской среде*. Если процедура будет запущена одновременно в двух транзакциях, они обе будут увеличивать один и тот же счетчик одновременно, так что результат будет бессмысленным. Хуже того, увеличение останавливающего генератора немедленно остановит процедуру в *обеих* транзакциях. Но, например, для месячных отчетов, которые генерируются единственным модулем в пакетном режиме, это может быть приемлемо - обычно, это зависит от ваших потребностей.

Если вы хотите использовать эту технику и позволять *пользователям* вызывать процедуру в любое время, вы должны убедиться другими средствами, что хранимая процедура не может быть запущена дважды. Подумав об этом, я предлагаю использовать для этого другой генератор: давайте назовем его gen_spTestLocked (предполагается, конечно же, что начальное значение равно 0):

```
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);
    /* Самый первый шаг: инкремент блокирующего генератора */

  if (lockcount = 1) then /* _мы_ получили блокировку, продолжаем */
  begin
    (..."обычное" тело процедуры...)
  end

  lockcount = GEN_ID(gen_spTestLocked, -1); /* выполняем декремент */

  /* Убеждаемся, что генератор сброшен в самом конце, когда исключение было
     выброшено внутри «обычного» тела процедуры: */

  WHEN ANY DO
    lockcount = GEN_ID(gen_spTestLocked, -1); /* декрементируем */
  exit;
END^
```

Примечание. Я не уверен на 100%, что это абсолютно безопасно для многопользовательской среды, но это позволит выполнить блокировку до тех пор, пока не произойдет ВЫХОД из тела процедуры, после чего процедура завершит свою работу, а до этого генератор остается инкрементированным. Выражение WHEN ANY обрабатывает исключения, но это не нормальный EXIT. При этом вы декрементируете генератор вручную - но вы должны декрементировать генератор непосредственно перед выходом (EXIT). Принимайте все меры предосторожности, я не могу воссоздать ситуацию, когда бы этот механизм не сработал... Если вы можете, дайте нам знать!

Приложение A: История документа

Полная история документа записана в модуле `manual` в нашем дереве CVS; смотрите http://sourceforge.net/cvs/?group_id=9028

История переиздания

0.1	4 апр 2006	FI	Первая редакция.
0.2	7 мая 2006	PV	Добавлен синтаксис SEQUENCE и другая информация относительно СУБД Firebird 2. Добавлена информация: важные замечания о клиентских диалектах; оператор SHOW GENERATOR и его коллеги; удаление генераторов и упаковка места хранения генераторов. Изменены и расширены следующие разделы: *Где хранятся генераторы?*, *Каково максимальное значение генератора?*, *Сколько генераторов...?*, *Использование операторов для генераторов*. Другие изменения, дополнения и корректировки различных разделов, в основном в первой половине документа. Небольшие изменения второй половины (начиная с *Использование генераторов для создания уникальных идентификаторов строк*).
0.2-ru	23 окт 2006	SK	Документ переведен на русский язык.
0.2-ru	29 окт 2006	PM	Корректировка перевода на русский язык.

Приложение B: Лицензионное соглашение

Оригинальное название документа: *Firebird Generator Guide*.

Автор исходного документа: Frank Ingermann.

Соавтор: Paul Vinkenoog – см. историю документа.

Перевод на русский язык: Сергей Ковалёв.

Корректор перевода: Павел Меньщиков.