

Um guia de como e quando utilizar Generators no Firebird



FRANK INGERMANN

Este artigo explica o que são os *Generators* do Firebird e como e por que devemos utilizá-los. Representa uma tentativa de coletar todas as informações relevantes a respeito de *Generators* em um único documento.

Quem deve ler este artigo?

Leia este artigo se você:

- Não está familiarizado com o conceito de *Generators*;
- Tem dúvidas de como utilizá-lo;
- Quer criar um campo ID no estilo “auto-incremento” como visto em outros banco de dados;
- Está procurando exemplos de como utilizar *Generatos* para incrementar campos;

O que é um generator?

Imaginemos um *Generator* como um contador de números inteiros residente em um banco de dados Firebird. Poderemos criar um *Generator* e ao mesmo tempo atribuir-lhe um nome, com a seguinte declaração:

```
CREATE GENERATOR GenTest;
```

A seguir podemos obter o seu valor atual, incrementá-lo ou decrementá-lo como faríamos com um *var i: integer* em Delphi, porém, não é fácil configurá-lo diretamente com certo valor de um modo previsível e a seguir obter o mesmo valor. Ele encontra-se no banco de dados, porém, fora do controle da transação.

O que é uma Sequence?

O termo *Sequence* é a designação oficial SQL para o que o Firebird chama de *Generator*. Como a equipe do Firebird está constantemente se esforçando para melhor se enquadrar no padrão SQL, o termo *SEQUENCE* pode ser utilizado como sinônimo de *GENERATOR* no Firebird 2 e superior. De fato, é recomendável a utilização da sintaxe *SEQUENCE*, quando formos escrever novo código.

Embora a palavra *Sequence* ponha a ênfase na série de valores gerados, ao passo que *Generator* parece referir-se principalmente ao mecanismo fonte que produz esses valores, não existe em absoluto nenhuma diferença entre um *Generator* e uma *Sequence*. São apenas duas palavras para nos referirmos ao mesmo recurso do banco de dados.

Podemos criar um *Generator* e acessá-lo utilizando a sintaxe de *Sequence* e vice-versa. A seguir, temos a sintaxe preferencial para criarmos um *Generator/Sequence* no Firebird 2:

```
CREATE SEQUENCE SeqTest;
```

Onde são armazenados os Generators?

As declarações dos *Generators* são armazenadas na tabela de sistema RDB\$GENERATORS. Os seus valores, no entanto, são armazenados em páginas reservadas especiais dentro do banco de dados. Nunca teremos acesso a esses valores diretamente, e sim por meio de funções e declarações embutidas que serão abordadas mais adiante.

Obs.: As informações fornecidas nessa seção têm objetivos educacionais somente. Como regra geral, não devemos “mexer” com as tabelas do sistema. Não devemos tentar criar ou alterar **Generators** escrevendo diretamente na tabela RDB\$GENERATORS (o acesso mediante declarações SELECT, no entanto, não tem nenhuma contra-indicação).

A estrutura da tabela de sistema RDB\$GENERATORS é a seguinte:

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

E ainda, no Firebird 2.0:

```
RDB$DESCRIPTION BLOB subtype TEXT
```

Observar que o GENERATOR_ID é, como o nome diz, um identificador para cada *Generator* e não o seu valor. Também, não devemos permitir que nossas aplicações armazenem o identificador do generator para utilização posterior. Além de não fazer sentido, o ID pode ser modificado após um ciclo de backup/restore. O SYSTEM_FLAG é 1 para *Generators* utilizados internamente pelo engine do servidor e NULL ou 0 para aqueles criados pelo usuário.

Agora daremos uma olhada na tabela RDB\$GENERATORS, aqui com um simples *Generator* auto-definido, conforme vemos na **Tabela 1**.

Notas para o Firebird 2:

- No Firebird 2 foi introduzido um *Generator* de sistema, chamado RDB\$BACKUP_HISTORY. É utilizado para a nova funcionalidade do NBackup;
- Embora a sintaxe SEQUENCE seja preferida, a tabela de sistema RDB\$GENERATORS e as suas colunas não foram renomeadas no Firebird 2.

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\$FIELD_NAME	6	1
RDB\$INDEX_NAME	7	1
RDB\$TRIGGER_NAME	8	1
MY_OWN_GENERATOR	9	NULL

Tabela 1. Estrutura da tabela RDB\$GENERATORS

Qual é o valor máximo de um Generator?

Os *Generators* armazenam e retornam valores de 64 bits em todas as versões do Firebird. Isso nos dá uma variação de valor de: $-2^{63}.. 2^{63-1}$ ou $-9.223.372.036.854.775.808.. 9.223.372.036.854.775.807$

Portanto, se utilizamos um *Generator* com o valor inicial 0 para alimentar uma coluna NUMERIC(18) ou BIGINT (ambos tipos representam números inteiros de 64 bits) e a seguir inserirmos 1000 linhas por segundo (o que já é uma frequência considerável), levaria aproximadamente 300 milhões de anos (!) até atingirmos o valor máximo. Como é bastante improvável que o ser humano ainda ande neste planeta até lá (e ainda estiver utilizando bancos de dados Firebird), não é nada com o que devamos nos preocupar.

No entanto, um alerta. O Firebird fala dois dialetos SQL: o dialeto 1 e o dialeto 3. Novos bancos de dados sempre devem ser criados com o dialeto 3, que é mais poderoso em vários aspectos. O dialeto 1 é para compatibilidade, deve ser utilizado somente para bancos de dados legados, que foram criados sob o InterBase 5.6 ou anterior.

Uma das diferenças entre os dois é que o dialeto 1 não tem disponível nenhum tipo numérico nativo de 64. As colunas NUMERIC(18), por exemplo, são totalmente armazenadas com DOUBLE PRECISION, que é do tipo ponto flutuante. O maior número do tipo inteiro no dialeto 1 é o INTEGER de 32 bits.

Tanto no dialeto 1 quanto no dialeto 3, os generators são de 64 bits. Porém, se atribuirmos os valores gerados a uma coluna INTEGER em um banco de dados gerado com o dialeto 1, esses valores serão truncados para os 32 bits de mais baixa ordem, produzindo uma variação eficaz de:

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

Embora o próprio *Generator* continue gerando valores de 2.147.483.647 a 2.147.483.648, o valor truncado ficaria em torno deste ponto, dando a impressão de um *Generator* de 32 bits. Na situação descrita anteriormente, com 1000 inserções por segundo, a coluna alimentada pelo *Generator* agora iria estourar após 25 dias (!!!), o que de fato é algo com o que se preocupar.

O valor 2^{31} é grande, porém, novamente nesse caso, não muito preocupante dependendo da situação.

Obs.: No dialeto 3, se atribuirmos valores **Generator** a

um campo *INTEGER*, tudo funcionará bem enquanto os valores permanecerem dentro da faixa de 32 bits. No entanto, assim que a faixa for excedida, obteremos um erro de overflow. O dialeto 3 é muito mais preciso na verificação de faixa do que o dialeto 1!

Dialeto cliente e valores de Generator

Os clientes que falam com um servidor Firebird, podem configurar o seu dialeto em 1 ou 3, independentemente do banco de dados que estejam conectados. É o dialeto do cliente e não o dialeto do banco de dados, que determina o modo como o Firebird passa valores de *Generators* para o cliente:

- Se o dialeto cliente for 1, o servidor retorna valores como números inteiros de 32 bits truncados para o cliente. No entanto, dentro do banco de dados os valores permanecem com 64 bits e não estouram após atingir 2^{31-1} (embora possam ter esse aspecto para o cliente). Isso é verdadeiro tanto para bancos de dados dialeto 1 quanto para o dialeto 3;
- Se o dialeto cliente for 3, o servidor passa o valor de 64 bits completo para o cliente.

Quantos Generators há disponíveis em um banco de dados?

Desde o Firebird versão 1.0, o número de *Generators* que poderemos ter em um único banco de dados é limitado somente pelo máximo ID designado na tabela do sistema RDB\$GENERATORS. Sendo um SMALLINT, o valor máximo será de 2^{15-1} ou 32767.

O primeiro ID é sempre 1, portanto, o número total de *Generators* não pode exceder 32767. Como discutido antes, existem 8 ou 9 *Generators* de sistema no banco de dados, deixando espaço para utilização pelo usuário de no mínimo 32758 *Generators*.

Isso deve ser amplo o bastante para atender às necessidades de qualquer aplicação prática e desde que o desempenho não é afetado pela quantidade de *Generators* declarados, estamos livres para utilizar tantos *Generators* quantos sejam necessários.

Versões mais antigas do InterBase e do Firebird

Nas versões anteriores à versão 1.0 do Firebird, bem como no InterBase, somente uma página do banco de dados era utilizada para armazenamento dos valores de *Generators*. Por isso, o número de *Generators* disponíveis estava limitado pelo tamanho da página do banco de dados.

A **Tabela 2** a seguir, demonstra quantos *Generators* (incluindo os *Generators* de sistema) podemos ter nas várias versões do InterBase e do Firebird (agradecemos ao Paul Reeves por ter nos fornecido as informações iniciais):

Em versões do InterBase anteriores à 6, os *Generators* tinham somente 32 bits de comprimento. Isso explica por que essas versões mais antigas podiam mal armazenar duas vezes o número de *Generators* no mesmo tamanho de página.

Versão	Tamanho da Página			
	1 K	2 K	4 K	8 K
InterBase <v.6	247	503	1015	2039
InterBase 6 e primeiras Firebird pré-1.0	123	251	507	1019
Todas as versões posteriores do Firebird	32767			

Tabela 2. Quantidade de Generators do banco de dados

Generators e transações

Como já foi dito, os *Generators* existem fora do controle das transações. Isso significa simplesmente que não podemos, com segurança, fazer o *rollback* dos *Generators* dentro de uma transação. Outras transações podem estar executando simultaneamente e modificando o valor enquanto a nossa transação está rodando. Portanto, uma vez que tivermos solicitado um valor de *Generator*, devemos considerá-lo como “não mais utilizável”.

Quando iniciamos uma transação e a seguir chamamos um *Generator* e obtemos um valor de, digamos 5, esse *Generator* permanecerá com esse valor mesmo se fizermos o *rollback* da transação (!). Nem vale a pena pensar em alguma coisa como “tudo bem, quando acontecer o *rollback*, podemos posteriormente apenas fazer *GEN_ID(mygen,-1)* para diminuí-lo para 4”.

Isso pode funcionar na maioria das vezes, porém, é perigoso, porque outras transações simultâneas podem ter modificado o valor no meio tempo. Pela mesma razão, não faz sentido obtermos o valor atual com *GEN_ID(mygen, 0)* e a seguir incrementar o valor no lado cliente.

Declarações SQL para Generators

O nome de um *Generator* deve ser um meta-identificador de banco de dados habitual: máximo de 31 caracteres, nenhum caractere especial exceto o underline “_” (a menos que utilizemos identificadores entre aspas duplas).

Os comandos e declarações SQL que se aplicam a *Generators*, estão enumerados a seguir:

Declarações DDL:

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

Declarações DML no cliente SQL:

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

Sintaxe recomendada para o Firebird 2

Embora a sintaxe tradicional ainda seja totalmente suportada no Firebird 2, esses são os DDLs equivalentes recomendados:

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

E para declarações DML:

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

Atualmente, a nova sintaxe não suporta um incremento diferente de 1. Essa limitação será eliminada nas versões futuras. Entretanto, utilize o GEN_ID quando for necessário aplicar outro valor de incremento.

Utilizando declarações para Generators

A disponibilidade de declarações e funções depende de onde elas são utilizadas: cliente SQL (conectado ao servidor) ou PSQL (linguagem utilizada em *Stored Procedures* e *Triggers*).

Criação de um Generator (Insert)

No cliente SQL:

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

Recomendado para o Firebird 2 e superior:

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

Em PSQL, não é possível, pois não podemos modificar os metadados do banco de dados em *Stored Procedures* ou *Triggers*, tampouco podemos criar *Generators*.

Obs.: No Firebird 1.5 e superior, podemos contornar esta limitação com as funcionalidades do EXECUTE STATEMENT.

Obtendo o valor atual (Select)

No cliente SQL:

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

Essa sintaxe é ainda a única opção para o Firebird 2.

Em PSQL:

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

No Firebird 2 usamos a mesma sintaxe.

Gerando o próximo valor (Update + Select)

Assim como fazemos para obter o valor atual (GEN_ID), agora utilizando um valor de incremento de 1. O Firebird irá:

1. obter o valor atual do *Generator*;
2. incrementá-lo de 1;
3. retornar o valor incrementado.

No cliente SQL:

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

A nova sintaxe, que é preferida para o Firebird 2, é inteiramente diferente:

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

Em PSQL:

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

Preferida para o Firebird 2 e superior:

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

Atenção

Na ferramenta isql de linha de comando do Firebird, existem dois comandos adicionais para recuperar o valor atual do Generator:

```
SHOW GENERATOR <GeneratorName>;
SHOW GENERATORS;
```

O primeiro comando recupera o valor atual do Generator especificado, enquanto que o segundo faz o mesmo para todos os Generators no banco de dados que não são do sistema. Os dois comandos equivalentes preferidos para o Firebird são, como poderíamos supor:

```
HOW SEQUENCE <SequenceName>;
SHOW SEQUENCES;
```

Note novamente que esses comandos (SHOW) estão somente disponíveis na ferramenta isql do Firebird. Diferentemente do GEN_ID, não podemos utilizá-los de dentro de outros clientes (a menos que esses clientes sejam front-ends isql).

Configurando um Generator diretamente para um determinado valor (Update)

No cliente SQL:

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

Isso é útil para pré-configurar *Generators* para um valor diferente de 0 (que é o valor padrão após a criação) em, por exemplo, um *script* para criar o banco de dados. Assim como o CREATE GENERATOR, essa é uma declaração DDL (não DML).

Sintaxe preferencial para o Firebird 2 e superior:

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

Em PSQL:

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

Obs.: Isso é mais do que um pequeno truque para fazer o que normalmente não poderíamos e não deveríamos fazer em **Stored Procedures** e **Triggers**: a configuração de **Generators**. Esses valores são obtidos e não configurados.

Eliminando um Generator (Delete)

No cliente SQL:

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

Preferido para o Firebird 2 e superior:

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

Em PSQL não é possível pois, novamente, não devemos modificar metadados em PSQL.

A eliminação de um *Generator* não libera o espaço por ele ocupado para ser utilizado por um novo *Generator*. Na prática isso raramente é problemático, porque a maior parte dos bancos

de dados não têm as dezenas de milhares de *Generators* que o Firebird permite, portanto existe espaço suficiente para mais.

No entanto, se o banco de dados realmente corre o risco de atingir o teto de 32767, poderemos liberar o espaço dos *Generators* não mais utilizados, executando um ciclo de *backup/restore*. Isso resultará na compactação da tabela RDB\$GENERATORS, redefinindo uma série contígua de IDs.

Dependendo da situação, o banco de dados restaurado também pode precisar de menos páginas para os valores *Generator*.

Utilizando Generators para criar ID de linhas únicos

Afinal, por que ID de linhas?

A resposta a essa pergunta iria muito além do alcance deste artigo. Se não houver nenhuma necessidade de ter um manipulador genérico e único para cada linha dentro em uma tabela ou não gostarmos da idéia de chaves sem sentido ou representativas em geral, podemos provavelmente pular essa seção.

Um para todos ou um para cada um?

Tudo bem, portanto, desejamos ID de linhas. Uma decisão fundamental e básica a ser tomada, consiste em determinar se utilizaremos um *Generator* único para todas as tabelas ou um *Generator* independente para cada tabela. Isso depende de nós, porém, devemos levar em conta as seguintes considerações.

Na abordagem “um para todas”:

- + precisamos somente de um *Generator* único para todos os IDs;
- + teremos um número inteiro que não somente identifica a linha na tabela, porém, no banco de dados inteiro;
- - teremos menos valores de ID possíveis por tabela (isso, na verdade, não deverá ser um problema com *Generators* de 64 bits);
- - logo teremos que lidar com valores de ID grandes mesmo em, por exemplo, busca em tabelas com apenas um número pequeno de registros;
- - provavelmente detectaremos lacunas em uma *Sequence* ID a nível de tabela, pois os valores *Generator* estarão espalhados por todas as tabelas.

Na abordagem “um para cada”:

- - teremos que criar um *Generator* para cada tabela com ID no banco de dados;
- - sempre necessitaremos da combinação de ID e nome da tabela para identificar unicamente qualquer linha em qualquer tabela;
- + teremos um contador de inserções simples e robusto por tabela;
- + teremos uma sequência cronológica por tabela: se encontrarmos uma falha na sequência de IDs de uma tabela, esta terá sido causada por um DELETE ou por um INSERT falho.

Poderíamos reutilizar os valores Generator?

Tecnicamente sim, porém não deveríamos. Nunca mesmo!

Isso não destruiria apenas a sequência cronológica de IDs (não poderemos mais determinar a idade de uma linha apenas observando o ID), mas quanto mais pensarmos a respeito disso, mais dores de cabeça teremos.

Além disso, é uma absoluta contradição em relação a todo o conceito de identificadores de linha únicos. Portanto, a menos que tenhamos boas razões para reutilizar valores de *Generators* e um mecanismo bem bolado para fazer isso funcionar com segurança em ambientes multi-usuário/multi-transação, não devemos fazer isto.

Generators de IDs ou campos de auto-incremento

Dado um registro recentemente inserido, um ID (no tocante a um número de série único) é facilmente conseguido com os *Generators* e com o *Trigger Before Insert*, como veremos nas subseções seguintes.

Começaremos supondo que temos uma tabela chamada *Test* com uma coluna de ID declarada como inteiro. O nome do *Generator* é GIDTEST.

Trigger Before Insert, versão 1:

```
CREATE TRIGGER TRGTTEST_BI_V1 FOR TESTE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  NEW.ID = GEN_ID(GIDTEST, 1);
END
```

Problemas com a versão 1 do *Trigger*: funciona muito bem, porém, também desperdiça um valor *Generator* em casos onde já existe um ID fornecido na declaração INSERT. Portanto, será mais eficiente atribuir um valor somente quando não existir nenhum no INSERT.

Trigger Before Insert, versão 2:

```
CREATE TRIGGER TRGTTEST_BI_V2 FOR TESTE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    BEGIN
      NEW.ID = GEN_ID(GIDTEST, 1);
    END
  END
END
```

Problemas com a versão 2 do *Trigger*: alguns componentes de acesso têm o mau hábito de auto-preencher todas as colunas em uma declaração INSERT. Aqueles que não foram explicitamente configurados terão designados valores padrão, normalmente 0 para colunas do tipo inteiro.

Nesse caso, a *Trigger* não funcionará: detectará que a coluna ID não tem o estado de NULL, porém, contém o valor 0, portanto não gerará um novo ID. Assim mesmo, poderíamos postar o registro, embora apenas um; o segundo; falhará.

De qualquer maneira, é bom banir o “0” como um valor de ID normal, para evitar qualquer confusão com NULL e 0. Poderíamos, por exemplo, utilizar uma linha especial com um ID 0 para armazenar um registro padrão em cada tabela.

Trigger Before Insert, versão 3:

```
CREATE TRIGGER TRGTTEST_BI_V3 FOR TESTE
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
END
```

Bem, agora que temos um *Trigger* de ID robusto e funcional, os parágrafos seguintes explicarão porque geralmente não precisaremos dele em absoluto. O problema básico com IDs atribuídos com *Before Insert* é que esses são gerados do lado servidor após termos enviado a declaração de inserção do lado cliente.

Isso claramente significa que não há nenhuma maneira segura de conhecer do lado cliente, que ID foi gerado para a linha que acabamos de inserir. Poderíamos recuperar o valor do *Generator* do lado cliente depois da inserção, porém, em ambientes multi-usuário, não poderemos, na verdade, ter certeza de que o que obtivemos é o ID da linha desejada (por causa da questão da transação).

No entanto, se obtivermos um novo valor *Generator* antes e postarmos a inserção com esse valor, poderemos simplesmente recuperar a linha com um *Select ... where ID = <genvalue>* para ver quais padrões foram aplicados ou se as colunas foram afetadas pelos *Triggers* de inserção.

Isso funciona especialmente bem, porque normalmente temos uma chave única na coluna de ID e esses são os mais rápidos índices que poderemos obter, são imbatíveis em seletividade e geralmente menores do que índices em colunas do tipo CHAR(n).

Podemos concluir que: devemos criar um *Trigger Before Insert* para tornar absolutamente garantido que cada linha obterá um ID único, mesmo quando nenhum valor de ID for fornecido do lado cliente nas declarações de inserção.

Se tivermos um banco de dados onde o código da nossa aplicação é a única fonte para a inserção de registros, poderíamos então omitir o *Trigger*, porém, nesse caso, deveríamos sempre obter um novo valor *Generator* do banco de dados, antes de

emitir a declaração de inserção e incluí-lo nessa.

O mesmo, naturalmente, será válido para inserções através de *Triggers* e *Stored Procedures*.

O que mais podemos fazer com Generators

Vejamos aqui algumas idéias de *Generators* personalizados, além daqueles para a geração de ID de linhas únicos.

Utilizando Generators para atribuir, por exemplo, números únicos para arquivos de transferência

Um uso clássico de *Generators* deve garantir números únicos, sequenciais para, bem, qualquer outra coisa na aplicação além dos ID de linhas tal qual discutido anteriormente. Se tivermos uma aplicação que está transferindo dados para algum outro sistema, poderíamos utilizar *Generators* para identificar com segurança uma única transferência, assinalando-a com um valor gerado.

Isso ajuda muito no rastreamento de problemas com interfaces entre dois sistemas (e, diferentemente da maior parte dos seguintes, isso realmente funciona com segurança e exatidão).

Generators como contadores de uso para que as Stored Procedures possam prover estatísticas básicas

Imaginemos que acabamos de incorporar uma nova funcionalidade fantástica ao banco de dados com uma *Stored Procedure*. Agora atualizamos os sistemas dos clientes e depois de algum tempo gostaríamos de saber se os usuários na verdade utilizam essa funcionalidade e com que frequência.

Simples: construímos um *Generator* especial que será incrementado somente na *Stored Procedure* e pronto. Apenas uma restrição: não poderemos conhecer a quantidade de transações para as quais houve *rollback* após ou enquanto a *Stored Procedure* executou.

No entanto, pelo menos saberemos com que frequência os usuários tentaram utilizar a nossa *Stored Procedure*. Poderíamos, além disso, refinar esse método utilizando dois *Generators*: cada um será incrementado logo no início da *Stored Procedure* e o outro precisamente antes do EXIT.

Dessa maneira, poderemos contar quantas tentativas de utilização da *Stored Procedure* foram bem sucedidas: se ambos os



Generators tiverem o mesmo valor, então nenhuma das chamadas à *Stored Procedure* falhou. Naturalmente, ainda não sabemos quantas vezes a(s) transação(ões) que invocam a nossa *Stored Procedure* de fato realizaram o *Commit*.

Generators para simular “Select count(*) from ...”

Existe um problema bem conhecido com o InterBase e com o Firebird, que diz respeito ao fato de que um SELECT COUNT(*) (sem a cláusula *where*) para uma tabela bem grande, pode demorar bastante para executar, pois o servidor deve contar na mão, quantas linhas existem na tabela no momento da solicitação.

Em teoria, poderíamos facilmente resolver esse problema com *Generators*:

- Criamos um *Generator* especial para contar linhas;
- Construímos um *Trigger Before Insert* que o incrementa;
- Construímos um *Trigger After Delete* que o decrementa.

Isso funciona muito bem e torna desnecessária uma contagem completa de registros: apenas obtém o valor atual do *Generator*. Realçamos aqui na teoria, porque tudo isso vai por água abaixo quando qualquer declaração de inserção falha, porque como foi mencionado antes, os *Generators* estão além do controle da transação. As inserções podem falhar por causa de restrições (violações de chave únicas, campos NOT NULL que são NULL etc.) ou outras restrições de metadados, ou simplesmente porque foi feito o *rollback* da transação que emitiu a inserção.

Poderíamos não ter nenhuma linha na tabela e mesmo assim, o contador de inserções aumentará. Portanto, depende: se conhecermos a porcentagem de inserções que falham (podemos ter uma idéia disso) e somente estamos interessados em uma estimativa da contagem de registros, nesse caso, esse método pode ser útil mesmo que não seja exato.

De vez em quando poderíamos fazer uma contagem de re-

Listagem 1. Exemplo de uso de Generator para monitorar uma SP

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;
SET TERM ^;
CREATE PROCEDURE spTest (...)
AS
BEGIN
  (...)
  FOR SCT <LOTS OF DATA TAKING LOTS OF TIME> DO
  BEGIN
    GEN_ID(GEN_SPTSTPROGRESS,1);
    IF (GEN_ID(GEN_SPTSTSTOP,0)>0) THEN EXIT;
    (...NORMAL PROCESSING HERE...)
  END
END^
```

Listagem 2. Usando Generator para monitorar uma SP (com lock)

```
CREATE GENERATOR GEN_SPTSTPROGRESS;
CREATE GENERATOR GEN_SPTSTSTOP;
CREATE GENERATOR GEN_SPTSTLOCKED;
SET TERM ^;
CREATE PROCEDURE SPTEST (...)
AS
DECLARE VARIABLE LOCKCOUNT INTEGER;
BEGIN
  LOCKCOUNT = GEN_ID(GEN_SPTSTLOCKED, 1);
  IF (LOCKCOUNT = 1) THEN
  BEGIN
    (CORPO DA PROCEDURE)
  END
  LOCKCOUNT = GEN_ID(GEN_SPTSTLOCKED, -1);
  WHEN ANY DO
    LOCKCOUNT = GEN_ID(SPTSTLOCKED, -1);
EXIT;
END^
```

gistros “normal” e configurar o *Generator* com o valor exato (re-sincronizar o *Generator*), para que o erro seja minimizado. Existem situações em que os clientes podem conviver com uma informação do tipo “existem *aproximadamente* 2,3 milhões de registros”, obtida de forma quase instantânea com um clique do mouse, porém, iria ficar muito insatisfeito se tiver que esperar 10 minutos ou mais para descobrir que existem precisamente 2.313.498.229 linhas...

Generators para monitorar e/ou controlar Stored Procedures demoradas

Stored Procedures que, por exemplo, geram saídas de relatório para grandes tabelas e/ou para *joins* complexos, podem demorar bastante para executar. Os *Generators* podem ser úteis aqui de duas maneiras: podem fornecer um “contador de progresso”, que, por sua vez, poderíamos consultar periodicamente do lado cliente enquanto a *Stored Procedure* está rodando e pode ser utilizado para pará-la (**Listagem 1**).

O código anterior representa apenas um rascunho, porém, dá para termos uma idéia. Do lado cliente, poderíamos emitir um *GEN_ID(gen_spTestProgress, 0)* de modo assíncrono com a extração da linha real (isso é, em uma *thread* diferente), para ver quantas linhas foram processadas e exibir esse valor em algum tipo de janela de progresso.

Poderíamos inclusive construir um *GEN_ID(gen_spTestStop, 1)* para cancelar de fora a *Stored Procedure* a qualquer momento. Embora isso possa ser muito prático, tem uma forte limitação: não é a prova de ambiente multi-usuário. Se a *Stored Procedure* rodar simultaneamente em duas transações, isso irá estragar o *Generator* de progresso, ambas irão incrementar o mesmo contador ao mesmo tempo, portanto, o resultado será inútil.

Além disso, incrementar o *Generator* de parada iria parar imediatamente a *Stored Procedure* em ambas as transações. No entanto, para relatórios mensais, por exemplo, que são gerados por um módulo único, isso pode ser aceitável, como de hábito, depende das nossas necessidades.

Se quisermos utilizar essa técnica e permitir aos usuários disparar a *Stored Procedure* a qualquer momento, deveremos garantir por algum meio, que a *Stored Procedure* não poderá ser rodada duas vezes. Pensando a respeito disso, tivemos a idéia de utilizar outro *Generator* para isso: chamado de *gen_spTestLocked* (assumindo, naturalmente, o valor inicial de 0), presente na **Listagem 2**.

A cláusula *WHEN ANY* trata as exceções, porém, não *EXITs* normais. Portanto, teríamos que decrementá-lo “na mão”. Poderíamos porém, decrementar o *Generator* precisamente antes do *EXIT* para evitar essa situação. Tomando as devidas precauções, achamos que não existe nenhuma situação em que esse mecanismo poderia falhar.

Conclusão

Vimos neste artigo todos os fundamentos relativos ao uso de *Generators* no Firebird, além de melhorias presentes na versão 2.0. Mostramos como o banco de dados armazena dados sobre *Generators*, como utilizá-los em aplicações (cliente e server) além de usos não-trivias para o uso de *Generators*. ■