

# Introdução a *Stored Procedures* e *Triggers* no Firebird

Por Bill Todd, Borland Developers Conference San Diego 2000

Traduzido e adaptado com autorização do autor por:

Alessandro Cunha Fernandes, Comunidade Firebird, Julho de 2002

Uma *stored procedure* é um programa escrito numa linguagem própria para *procedures* e *triggers* do Firebird que é armazenado como parte do banco de dados. *Stored procedures* podem ser chamadas por aplicações cliente ou por outras *stored procedures* ou *triggers*. *Triggers* são quase a mesma coisa que *stored procedures* exceto pelo modo como são chamadas. *Triggers* são chamadas automaticamente quando uma alteração em uma linha da tabela ocorre. Este artigo examina primeiro as *stored procedures* e logo depois as *triggers*. Como você poderá ver a maioria das coisas que serão ditas sobre *stored procedures* se aplicarão também às *triggers*.

## Vantagens do uso de *Stored Procedures*

A maior vantagem do uso de *stored procedures* é a redução de tráfego na rede. Já que as *stored procedures* são executadas pelo Firebird na máquina servidora de banco de dados, você pode utiliza-las para mover grande parte do seu código de manipulação de dados para o servidor. Isto elimina a transferência de dados do servidor para o cliente, pela rede, para a manipulação e reduzir tráfego é aumentar performance, particularmente em uma WAN ou em qualquer conexão de baixa velocidade.

*Stored procedures* aumentam a performance de outra forma também. Você pode utilizar queries para fazer muitas das coisas que podem ser feitas com *stored procedures* mas uma query tem uma grande desvantagem. Cada vez que a aplicação cliente envia um comando SQL para o servidor o comando tem que ser "*parsed*", ou seja, analisado gramaticalmente, submetido ao otimizador para formulação de um plano de execução. *Stored procedures* são analisadas, optimizadas e armazenadas em uma forma executável no momento em que são adicionadas ao banco de dados. A partir do momento que uma *stored procedure* não tem que ser analisada e optimizada cada vez que é chamada, ela é executada mais rapidamente que uma query equivalente. *Stored procedures* podem também executar operações muito mais complexas que uma simples query.

Se mais de uma aplicação irá acessar o banco de dados, as *stored procedures* podem também economizar tempo de manutenção e desenvolvimento já que qualquer aplicação poderá chama-la. A manutenção é mais fácil porque você pode alterar a *stored procedure* sem ter que alterar ou mesmo recompilar cada aplicação cliente.

Finalmente, *stored procedures* tem uma grande importância na segurança do banco de dados uma vez que elas podem acessar tabelas que o usuário não tem o direito de fazê-lo. Por exemplo, suponha que um usuário precise rodar um relatório que mostra o total de salários por departamento e nível salarial. Embora estas informações venham da tabela de salários dos empregados você não quer que este usuário tenha acesso aos salários individuais de todos os empregados. A solução é escrever uma *stored procedure* que extraia da tabela de salários as informações resumidas que o relatório precisa e dar direitos de

leitura à *stored procedure* para a tabela de salários. Você pode então dar direito ao usuário de executar a *stored procedure*. O usuário não precisa ter direitos sobre a tabela de salários.

## Quando você deve usar *Stored Procedures*?

A resposta curta é, sempre que você puder. Não existem desvantagens em se usar *stored procedures*. Existem apenas duas limitações. Primeiro, você tem que ser capaz de passar qualquer informação variável para a *stored procedure* como parâmetros ou coloca-las em uma tabela que a *stored procedure* possa acessar. Segundo, a linguagem de escrita de *stored procedures* e *triggers* pode ser muito limitada para operações mais complexas.

## Usando o comando CREATE PROCEDURE

*Stored procedures* são criadas através do comando CREATE PROCEDURE que tem a seguinte sintaxe:

```
CREATE PROCEDURE NomedProcedure
<parâmetros de entrada>
RETURNS
<parâmetros de saída>
AS
<declaração de variáveis locais>
BEGIN
<comandos da procedures>
END
```

Os parâmetros de entrada permitem à aplicação cliente passar os valores que serão usados para modificar o comportamento da *stored procedure*. Por exemplo, se o objetivo da *stored procedure* é calcular o total mensal da folha de pagamento para a um determinado departamento, o número do departamento deverá ser passado para a *stored procedure* como um parâmetro de entrada. Parâmetros de saída ou de retorno são é o meio pelo qual a *stored procedure* retorna informações para a aplicação cliente. Em nosso exemplo, o total da folha de pagamento mensal para o departamento passado deverá ser retornado em um parâmetro de saída. Um parâmetro pode ser de qualquer tipo de dados do Firebird exceto BLOB ou ARRAY. A *procedure* a seguir demonstra o uso tanto dos parâmetros de entrada como os de saída:

```
CREATE PROCEDURE SUB_TOT_BUDGET(
HEAD_DEPT CHAR(3)
)
RETURNS (
TOT_BUDGET NUMERIC (15, 2),
AVG_BUDGET NUMERIC (15, 2),
MIN_BUDGET NUMERIC (15, 2),
MAX_BUDGET NUMERIC (15, 2)
)
AS
BEGIN
SELECT SUM(BUDGET),
AVG(budget), MIN(budget), MAX(budget)
FROM department
WHERE head_dept = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
SUSPEND;
END ^
```

Esta *stored procedure* declara um parâmetro de entrada, HEAD\_DEPT cujo tipo é CHAR(3) e quatro parâmetros de saída, TOT\_BUDGET, AVG\_BUDGET, MIN\_BUDGET, e MAX\_BUDGET todos do tipo NUMERIC(15, 2). Tanto os parâmetros de entrada quanto os de saída devem estar entre parênteses. O comando SUSPEND pausa a *stored procedure* até que o cliente busque os valores dos parâmetros de saída. Este comando é explicado em mais detalhes mais tarde neste mesmo artigo.

## Declarando variáveis locais

Você pode declarar variáveis locais de qualquer tipo suportado pelo Firebird dentro de uma *stored procedure*. Estas variáveis só existem enquanto a *stored procedure* está sendo executada e seu escopo é local à *procedure*. Note que não existem variáveis globais quando se trabalha com *stored procedures* e *triggers* e elas não são necessárias. Se você tem valores que precisam ser compartilhados por duas ou mais *procedures*, você pode passá-los por parâmetros ou guardá-los em uma tabela.

Variáveis locais são declaradas depois da palavra chave AS e antes da palavra chave BEGIN que identifica o início do corpo da *stored procedure*. Para declarar variáveis use DECLARE VARIABLE <Nome variável> <Tipo variável>

```
DECLARE VARIABLE OrderCount Integer;  
DECLARE VARIABLE TotalAmount NUMERIC(15,2);
```

Note que cada comando DECLARE VARIABLE só pode declarar uma variável. A *procedure* a seguir ilustra o uso do comando DECLARE VARIABLE. Ela declara quatro variáveis locais, ord\_stat, hold\_stat, cust\_no and any\_po. Observe que quando uma variável é usada na cláusula INTO de um comando SELECT um sinal de dois pontos ':' deve ser adicionado como primeiro caracter do nome da variável, entretanto quando a variável é usada em qualquer outra parte este sinal não é mais necessário.

```
CREATE PROCEDURE SHIP_ORDER(  
PO_NUM CHAR(8)  
)  
AS  
DECLARE VARIABLE      ord_stat CHAR(7);  
DECLARE VARIABLE      hold_stat CHAR(1);  
DECLARE VARIABLE      cust_no INTEGER;  
DECLARE VARIABLE      any_po CHAR(8);  
BEGIN  
SELECT s.order_status, c.on_hold, c.cust_no  
FROM sales      s, customer c  
WHERE po_number = :po_num  
AND s.cust_no   = c.cust_no  
INTO :ord_stat, :hold_stat, :cust_no;  
  
/* Este pedido já foi enviado */  
IF (ord_stat    = 'shipped') THEN  
BEGIN  
EXCEPTION      order_already_shipped;  
SUSPEND;  
END  
  
/* Cliente está em atraso. */  
ELSE IF (hold_stat    = '*') THEN  
BEGIN  
EXCEPTION      customer_on_hold;  
SUSPEND;
```

```

END

/*
* Se existe uma conta não paga de pedidos enviados a mais de 2 meses,
* passe o cliente para cliente em atraso.
*/
FOR SELECT      po_number
FROM sales
WHERE cust_no   = :cust_no
AND order_status = 'shipped'
AND paid =      'n'
AND ship_date  < CAST('NOW' AS DATE) - 60
INTO :any_po
DO
BEGIN
EXCEPTION      customer_check;

UPDATE customer
SET on_hold    = '*'
WHERE cust_no  = :cust_no;
SUSPEND;
END

/*
* Envia o pedido.
*/
UPDATE sales
SET order_status = 'shipped', ship_date = 'NOW'
WHERE po_number = :po_num;
SUSPEND;
END ^

```

## Escrevendo o corpo da *procedure*

O corpo da *stored procedure* consiste em um conjunto de qualquer número de comandos da linguagem de escrita de *stored procedure* e *triggers* do Firebird dentro de um bloco BEGIN/END. O corpo da seguinte *procedure* consiste em um comando SELECT e um SUSPEND entre as palavras chave BEGIN e AND.

```

CREATE PROCEDURE SUB_TOT_BUDGET(
HEAD_DEPT CHAR(3)
)
RETURNS (
TOT_BUDGET NUMERIC (15,2),
AVG_BUDGET NUMERIC (15,2),
MIN_BUDGET NUMERIC (15,2),
MAX_BUDGET NUMERIC (15,2)
)
AS
BEGIN
SELECT SUM(budget), AVG(budget), MIN(budget), MAX(budget)
FROM department
WHERE head_dept=:head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
SUSPEND;
END ^

```

Cada comando no corpo de uma *procedure* tem que terminar com um ponto e virgula ';'.

## Outros elementos de linguagem

A linguagem de escrita de *stored procedure* e *triggers* do Firebird inclui todas as construções de uma linguagem de programação estruturada assim como declarações próprias para trabalhar com dados em tabelas. A seguinte seção descreverá estes elementos.

### Comentários

Você pode colocar comentários onde quiser em uma *stored procedure* usando a sintaxe **/\* Este é um comentário \*/**. Um comentário pode ter várias linhas, mas comentários aninhados não são permitidos.

### Bloco de comandos (BEGIN-END)

A linguagem de *stored procedures* e *triggers* se assemelha ao Pascal em algumas construções como IF-THEN-ELSE e loops WHILE que somente podem conter um comando. Entretanto, as palavras chave BEGIN e END podem ser usadas para agrupar uma série de comandos de forma que eles se tornem um comando composto. Nunca coloque um ponto-e-vírgula após um BEGIN ou um END.

### Comandos de atribuição

A linguagem de *procedures* e *triggers* suporta comandos de atribuição da seguinte forma:

```
Var1 = Var2 * Var3;
```

Var1 tanto pode ser uma variável local quanto um parâmetro de saída. Var2 e Var3 tanto podem ser variáveis locais como parâmetros de entrada. A expressão à direita do sinal de igual pode ser tão complexa quanto você deseje e você pode usar parênteses para agrupar operações com quantos níveis quiser.

### IF-THEN-ELSE

A sintaxe do comando IF no Firebird é a seguinte:

```
IF <expressão condicional> THEN  
<comando>  
ELSE  
<comando>
```

Onde <comando> pode ser tanto um comando simples quanto um bloco de comandos delimitado por um BEGIN-END. Na <expressão condicional> além dos operadores lógicos normais (=, <, >, <=, >=, <>) você pode usar também os seguintes operadores SQL:

Expressão Condicional	Descrição
Valor BETWEEN valor AND valor	Faixa de valores

Valor LIKE valor	O valor à direita pode incluir um ou mais curingas. Use % para zero ou mais caracteres e _ para um caracter.
Valor IN (valor1, valor2, valor3, &)	Membro de uma lista de valores.
Valor EXISTS (subquery)	Verdadeiro se o valor combinar com um dos valores retornados pela subquery.
Valor ANY (subquery)	Verdadeiro se o valor combinar com qualquer das linhas retornadas pela subquery.
Valor ALL (subquery)	Verdadeiro se o valor combinar com todas as linhas retornadas pela subquery.
Valor IS NULL	Verdadeiro se o valor for nulo.
Valor IS NOT NULL	Verdadeiro se o valor não for nulo.
Valor CONTAINING valor	Busca de <i>substring</i> sem diferenciar maiúsculas e minúsculas.
Valor STARTING WITH valor	Verdadeiro se o valor a esquerda iniciar com o valor a direita. Diferencia maiúsculas e minúsculas.

Exemplos de comandos IF válidos são:

```
IF
(any_sales > 0) THEN
BEGIN
EXCEPTION reassign_sales;
SUSPEND;
END

IF
(first IS NOT NULL) THEN
line2=first || ' ' || last;
ELSE
line2=last;
```

Note no exemplo acima que na linguagem de escrita de *stored procedures* e *triggers* no Firebird o operador de concatenação de strings é || (Duas barras verticais) e não o + como acontece na maioria das linguagens de programação.

```
IF (:mngr_no IS NULL) THEN
BEGIN
mngr_name='--TBH--';
title='';
END
ELSE
SELECT full_name, job_code
FROM employee
WHERE emp_no=:mngr_no
INTO :mngr_name, :title;
```

## WHILE-DO

A estrutura WHILE-DO permite criar loops nas *stored procedures* e *triggers*. A sintaxe é:

```
WHILE (<expressão condicional>) DO <comando>
```

Onde <comando> pode ser um bloco de comandos delimitado por um par BEGIN-END. Observe que a expressão condicional tem que estar entre parênteses.

```
WHILE (i <=5) DO
BEGIN
SELECT language_req[:i] FROM job
WHERE ((job_code=:code) AND (job_grade=:grade) AND (job_country=:cty)
AND (language_req IS NOT NULL))
INTO :languages;
IF (languages=' ')THEN /* Imprime 'NULL' ao invés de espaços */
languages='NULL';
i=i +1;
SUSPEND;
END
```

## Usando comandos SQL nas STORED PROCEDURES

Você pode usar os comandos SQL SELECT, INSERT, UPDATE e DELETE em uma *stored procedure* exatamente como você faria em uma query apenas com algumas pequenas alterações na sintaxe. Para todos esses comandos você pode usar variáveis locais ou parâmetros de entrada em qualquer lugar que um valor literal seria aceito. Por exemplo, no comando INSERT, a seguir, os valores inseridos são obtidos de um parâmetro de entrada.

```

CREATE PROCEDURE ADD_EMP_PROJ(
EMP_NO SMALLINT,
PROJ_ID CHAR(5)
)
AS
BEGIN
BEGIN
INSERT INTO employee_project (emp_no, proj_id) VALUES (:emp_no, :proj_id);
WHEN SQLCODE -530 DO
EXCEPTION unknown_emp_id;
END
SUSPEND;
END ^

```

A Segunda diferença é a adição da cláusula INTO ao comando SELECT de modo que você possa selecionar valores diretamente para variáveis ou parâmetros de saída como mostrado no exemplo a seguir:

```

CREATE PROCEDURE CUSTOMER_COUNT
RETURNS (
CUSTOMERCOUNT INTEGER
)
AS
BEGIN
SELECT COUNT(*) FROM CUSTOMER INTO :CustomerCount;
SUSPEND;
END ^

```

Você não pode usar comandos SQL DDL em uma *stored procedure*. Esta restrição se aplica aos comandos CREATE, ALTER, DROP, SET, GRANT, REVOKE, COMMIT e ROLLBACK.

### Usando FOR SELECT e DO

O exemplo anterior do comando SELECT que seleciona um ou mais valores para uma variável é bom desde que o SELECT retorne apenas uma linha. Quando precisar processar varias linhas retornadas por um SELECT você deverá usar o comando FOR SELECT e DO como mostrado a seguir:

```

CREATE PROCEDURE ORDER_LIST(
CUST_NO INTEGER
)
RETURNS (
PO_NUMBER CHAR(8)
)
AS
BEGIN
FOR SELECT PO_NUMBER FROM SALES
WHERE CUST_NO=:CUST_NO
INTO :PO_NUMBER
DO
SUSPEND;
END ^

```

Esta *procedure* pega um código de cliente de seu parâmetro de entrada e retorna os números de todas as compras do cliente da tabela SALES (vendas). Observe que os

números das vendas são todos retornados através de uma única variável de saída. Veja como isso funciona: A palavra chave FOR diz para o Firebird abrir um cursor no conjunto de resultados (result set) do comando SELECT. O comando SELECT tem que incluir a cláusula INTO que atribui cada campo retornado pelo SELECT a uma variável local ou parâmetro de saída. O comando após a palavra chave DO é executado para cada linha retornada pelo SELECT. O comando após o DO pode ser um bloco de comandos delimitado por um BEGIN-END.

### Usando o SUSPEND

No exemplo acima, o comando SUSPEND diz para a *stored procedure* suspender a execução até que uma solicitação de dados (fetch) seja recebida do cliente então a *procedure* lê o primeiro PO\_NUMBER para o parâmetro de saída e o retorna para o cliente. Cada vez que o cliente emite uma requisição, o próximo PO\_NUMBER é lido para o parâmetro de saída e retornado para o cliente. Isso continua até que todas as linhas retornadas pelo SELECT tenham sido processadas. SUSPEND não é só usado com FOR SELECT. Ele é usado sempre que a *stored procedure* retorna um valor para o cliente evitando que a *stored procedure* termine antes que o cliente tenha pego o resultado. A seguir um exemplo muito simples de uma *procedure* que retorna um valor em um parâmetro de saída:

```
CREATE PROCEDURE CUSTOMER_COUNT
RETURNS (
CUSTOMERCOUNT INTEGER
)
AS
BEGIN
SELECT COUNT(*) FROM CUSTOMER INTO :CustomerCount;
SUSPEND;
END ^
```

### Criando e modificando *Stored Procedures*

O método normal de se criar um banco de dados e seus objetos no Firebird é criar um script SQL no IBConsole ou em um editor de textos e então usar o IBConsole para executá-lo. Isto cria um problema já que tanto o IBConsole como a *stored procedure* usam o ponto-e-vírgula para terminar um comando.

### Lidando com o dilema do ponto-e-vírgula

O IBConsole identifica o fim de cada comando em um script SQL pelo caracter de término de comando que é por default o ponto-e-vírgula. Isto funciona bem na maioria dos casos mas não na criação de *stored procedures* ou *triggers*. O problema é que queremos que o IBConsole execute o comando CREATE PROCEDURE como um único comando e isso significa que ele deveria terminar com um ponto-e-vírgula. Entretanto, cada um dos comandos no corpo da *procedure* que se está criando também termina com um ponto-e-vírgula e o IBConsole acha que encontrou o fim do CREATE PROCEDURE quando ele encontra o primeiro ponto-e-vírgula. A única solução é trocar o caracter de término, que o IBConsole procura para identificar o fim do comando, por um outro diferente do ponto-e-vírgula. Para isso usamos o comando SET TERM. O script a seguir demonstra como:

```

SET TERM ^ ;
CREATE PROCEDURE Customer_Count
RETURNS (
CustomerCount Integer
)
AS
BEGIN
SELECT COUNT(*) FROM CUSTOMER
INTO :CustomerCount;
SUSPEND;
END ^
SET TERM ; ^

```

O primeiro comando SET TERM altera o caracter de término de comando para o caracter (^). Note que este comando ainda tem que terminar com um ponto-e-vírgula já que este ainda será o caracter de término até que o SET TERM ^ seja executado. O IBConsole irá agora ignorar os ponto-e-vírgula no final dos comandos no corpo da *procedures*. Um (^) é colocado logo após o END final no comando CREATE PROCEDURE. Quando o IBConsole encontra este caracter ele processa todo o comando CREATE PROCEDURE. O último SET TERM volta o terminador para o ponto-e-vírgula.

### Apagando e alterando *Stored Procedures*

Para remover uma *stored procedure* use o comando DROP PROCEDURE como a seguir:

```
DROP PROCEDURE Nome_Procedure;
```

Somente o SYSDBA ou o proprietário da *procedure* podem apaga-la. Use o comando ALTER PROCEDURE para alterar uma *stored procedure*. ALTER PROCEDURE tem exatamente a mesma sintaxe do comando CREATE PROCEDURE, apenas trocando a palavra CREATE por ALTER. A primeira vista pode parecer que você não precise do comando ALTER PROCEDURE já que você pode deletar a *stored procedure* e depois cria-la novamente com as alterações necessárias. Entretanto, isto não irá funcionar se a *procedure* que você está tentando alterar é chamada por outra *stored procedure*. Se a *stored procedure* 1 chama a *stored procedure* 2 você não pode apagar a *stored procedure* 2 porque a *stored procedure* 1 depende de sua existência.

Se você usar o IBConsole para exibir o *metadata* de seu banco de dados e examinar o código que cria a *stored procedure* você verá que o Firebird primeiro cria todas as *procedures* com o corpo vazio como mostrado no exemplo abaixo:

```

CREATE PROCEDURE ADD_EMP_PROJ (
EMP_NO SMALLINT,
PROJ_ID CHAR(5)
)
AS
BEGIN EXIT;END ^
CREATE PROCEDURE ALL_LANGS
RETURNS (
CODE VARCHAR(5),
GRADE VARCHAR(5),
COUNTRY VARCHAR(15),
LANG VARCHAR(15)
)
AS
BEGIN
EXIT;
END ^

```

Depois de todas as *procedures* terem sido criadas o script criado pelo Firebird usa o comando ALTER PROCEDURE para adicionar o corpo de cada *stored procedure*. Por exemplo:

```
ALTER PROCEDURE ADD_EMP_PROJ(  
EMP_NO SMALLINT,  
PROJ_ID CHAR(5)  
)  
AS  
BEGIN  
BEGIN  
INSERT INTO employee_project (emp_no, proj_id) VALUES (:emp_no, :proj_id);  
WHEN SQLCODE -530 DO  
EXCEPTION unknown_emp_id;  
END  
SUSPEND;  
END ^
```

Fazendo isto o Firebird elimina qualquer dependência entre as *procedures* quando elas estão sendo criadas. Já que o corpo de toda *procedure* esta vazio não pode haver dependência de *procedures* chamando outra(s). Quando os comandos ALTER PROCEDURE são executados eles podem ser rodados em qualquer ordem porque a declaração de qualquer *procedure* que eventualmente seja chamada pela *que* está sendo alterada no momento, já existirá.

### Chamando *Stored Procedures*

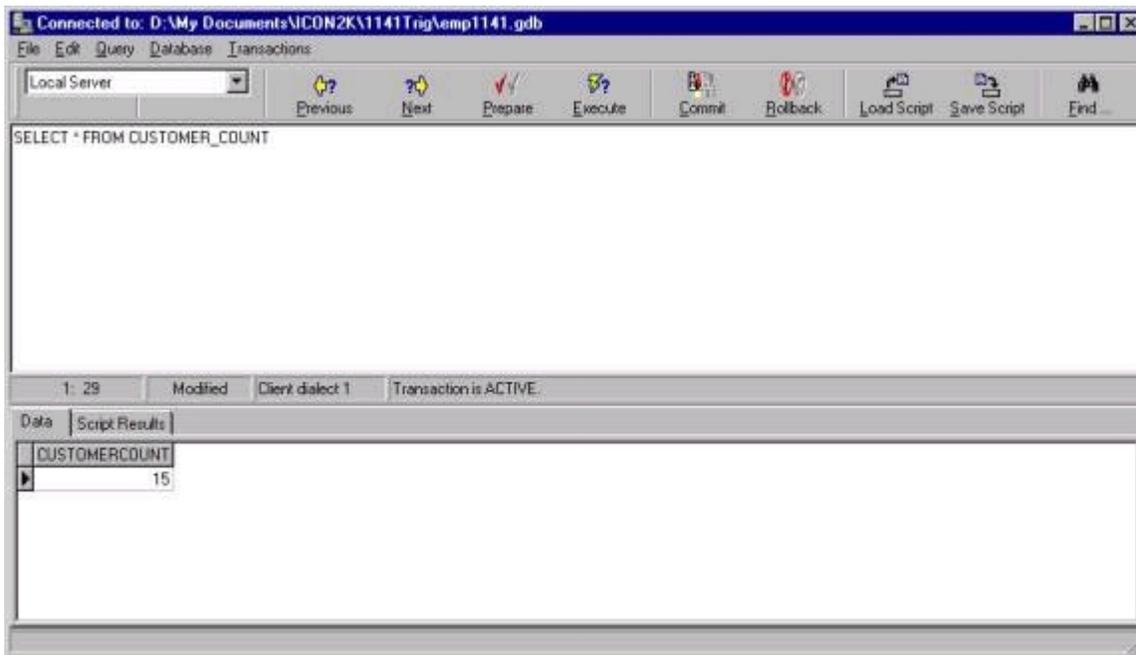
Você pode chamar *stored procedures* de outras *stored procedures* ou *triggers*, do IB Console ou de suas aplicações. *Stored procedures* no Firebird são divididas em dois grupos de acordo com como são chamadas. *Procedures* que retornam valores através de parâmetros de saída são chamadas de "*select procedures*" porque elas podem ser usadas no lugar de um nome de tabela em um comando SELECT.

### Chamando "*Select Procedures*"

"*Select procedures*" atribuem valores a parâmetros de saída e então executam um SUSPEND para retornar este valores. Abaixo, um exemplo simples de "*select procedure*".

```
CREATE PROCEDURE CUSTOMER_COUNT  
RETURNS (  
CUSTOMERCOUNT INTEGER  
)  
AS  
BEGIN  
SELECT COUNT(*) FROM CUSTOMER INTO :CustomerCount;  
SUSPEND;  
END ^
```

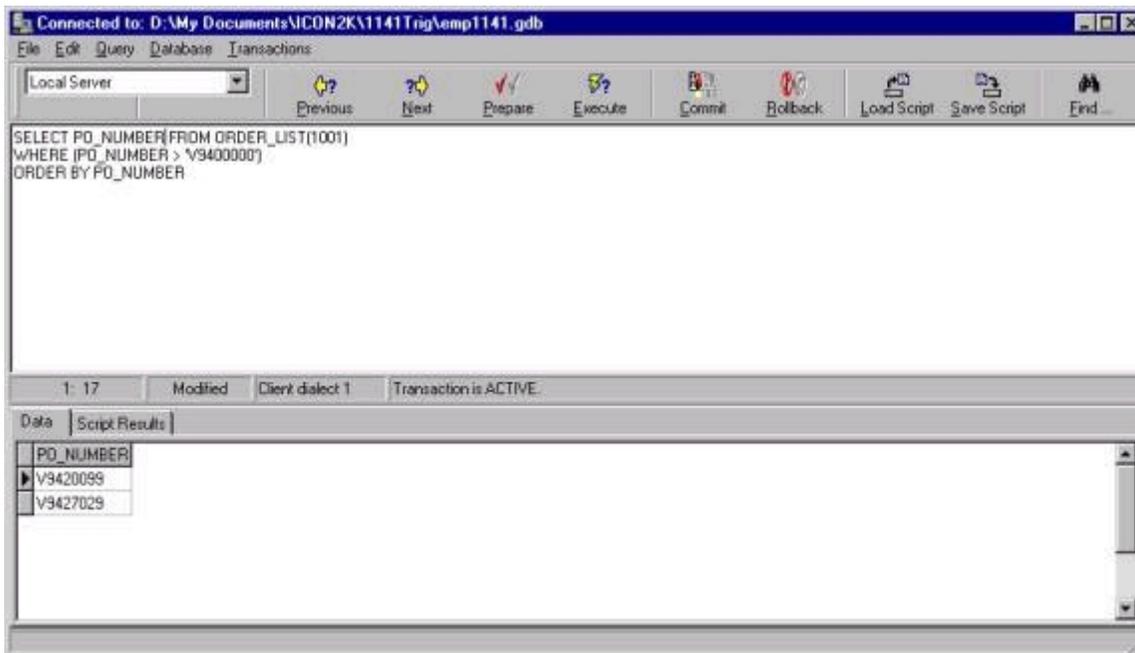
A figura 1 abaixo mostra esta *procedure* sendo chamada a partir do IB Console usando um comando SELECT. Observe que uma linha e uma coluna foram retornadas e o nome da coluna é o nome do parâmetro de saída.



**Figure 1 - Chamando uma *select procedure* a partir do IB Console**

Você também pode chamar *procedures* que necessitem de parâmetros de entrada a partir do IB Console. A figura 2 mostra um exemplo de chamada da seguinte *procedure*:

```
ALTER PROCEDURE ORDER_LIST(  
CUST_NO INTEGER  
)  
RETURNS (  
PO_NUMBER CHAR(8)  
)  
AS  
BEGIN  
FOR SELECT PO_NUMBER FROM SALES WHERE CUST_NO=:CUST_NO INTO :PO_NUMBER  
DO SUSPEND;  
END ^
```



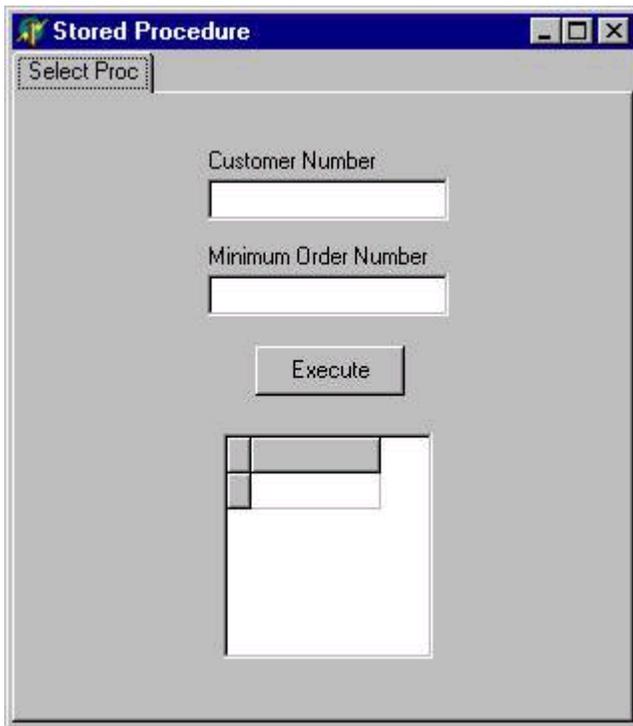
**Figure 2 - Executando uma *select procedure* com parâmetros de entrada**

O parâmetro de entrada, CUST\_NO, é passado para a *procedures* entre parênteses logo após o nome da *procedure* no comando SELECT. Note também que este comando SELECT inclui uma cláusula WHERE e uma cláusula ORDER BY. Isso permite que você chame uma *stored procedure* e retorne um subconjunto das linhas e colunas ordenados da forma que você desejar. Na verdade você pode tratar uma *select procedure* exatamente como uma tabela usando todas as possibilidades de um comando SELECT para controle do resultado que você obtém.

Você usará exatamente a mesma sintaxe para chamar uma *select procedure* a partir de outra *stored procedure*, de uma *trigger* ou de sua aplicação. O comando SQL a seguir foi retirado da propriedade SQL de um componente IBQuery de uma aplicação exemplo. A única diferença aqui é que os parâmetros :CUST\_NO e :PO\_NUMBER são usados para suprir os valores dos parâmetros de entrada.

```
SELECT * FROM ORDER_LIST(:CUST_NO)
WHERE (PO_NUMBER > :PO_NUMBER)
ORDER BY PO_NUMBER DESC
```

A figura 3 mostra o formulário usado para coletar o dado de entrada do usuário e executar a *stored procedure*.



**Figura 3 – A aplicação exemplo que executa a *select procedure***

O código a seguir foi retirado do evento “onclik” do botão “Execute” do formulário acima e mostra como os valores são atribuídos aos parâmetros de entrada antes da execução da *stored procedure*.

```
procedure TProcForm.SelectBtnClick(Sender: TObject);
begin
  with ProcDm.OrderListQry do
    begin
      Params.ParamByName('CUST_NO').Value := CustomerNoEdit.Text;
      Params.ParamByName('PO_NUMBER').Value := OrderNoEdit.Text;
      Open;
    end; //with
  end;
end;
```

### **Chamando uma “*Non-Select Procedure*”**

A *stored procedure* a seguir é um exemplo de uma “*non-select procedure*”, que é uma *procedure* que não retorna qualquer resultado. Esta *procedure* tem apenas um parâmetro de entrada, FACTOR, e ajusta os salários mínimo e máximo na tabela JOB por este fator.

```
CREATE PROCEDURE ADJUST_SALARY_RANGE(
  FACTOR FLOAT
)
AS
BEGIN
  UPDATE JOB
  SET MIN_SALARY=MIN_SALARY * :FACTOR, MAX_SALARY=MAX_SALARY * :FACTOR;
END ^
```

Use o comando EXECUTE PROCEDURE para rodar esta *stored procedure* a partir de uma *trigger*, outra *stored procedure* ou do IB Console. Por exemplo:

```
EXECUTE PROCEDURE ADJUST_SALARY_RANGE(1.1);
```

Para executar essa *stored procedure* a partir de sua aplicação use um componente `IBStoredProc` e o seguinte código:

```
with ProcDm.AdjustSalRngProc do
begin
  Params.ParamByName('Factor').Value := StrToFloat(FactorEdit.Text);
  Prepare;
  ExecProc;
end; //with
```

Em tempo de projeto sete a propriedade “Database” do componente `IBStoredProc` para o componente `IBDatabase` referente ao BD que contém a *stored procedure*. Sete a propriedade `StoredProcName` para o nome da *stored procedure* que você quer executar. Use o “Property editor” da propriedade “Params” para criar qualquer parâmetro de entrada necessário e configure seus tipos e valores default.

## Triggers

*Triggers* são iguais a *stored procedures* com as seguintes exceções:

1. *Triggers* são chamadas automaticamente quando os dados da tabela a qual ela esta conectada são alterados
2. *Triggers* não tem parâmetros de entrada.
3. *Triggers* não retornam valores.
4. *Triggers* são criadas pelo comando `CREATE TRIGGER`.

### Usando `CREATE TRIGGER`

O comando `CREATE TRIGGER`, a seguir, mostra todos os elementos da sintaxe do comando. As palavras-chave `CREATE TRIGGER` são seguidas do nome da *trigger*, a seguir a palavra-chave `FOR` e então o nome da tabela a qual a *trigger* estará relacionada. Em seguida vem a palavra `ACTIVE` (ativa) ou `INACTIVE` (inativa) indicando se a *trigger* deverá ou não ser executada. Se a *trigger* está inativa ela não será executada. Você verá como ativar e desativar uma *trigger* mais tarde neste artigo. O próximo elemento do comando `CREATE TRIGGER` indica quando a *trigger* será executada.

1. `BEFORE UPDATE` (Antes de uma atualização)
2. `AFTER UPDATE` (Após uma atualização)
3. `BEFORE INSERT` (Antes de uma inclusão)
4. `AFTER INSERT` (Após uma inclusão)
5. `BEFORE DELETE` (Antes de uma exclusão)
6. `AFTER DELETE` (Após uma exclusão)

A seguir vem a palavra chave opcional `POSITION` seguida de um número inteiro. O Firebird permite que você conecte quantas *trigger* quiser ao mesmo evento. Por exemplo, você poderia ter quatro *triggers* ligadas a tabela `EMPLOYEE` todas como `AFTER UPDATE`. Esta é uma grande característica já que permite que você modularize seu código. Entretanto a ordem em que as *trigger* vão ser executadas pode eventualmente ser importante. A palavra chave `POSITION` te dá o controle da ordem de execução baseado no inteiro informado. No exemplo abaixo a *trigger* mostrada será executada primeiro porque a sua posição é 0 (zero). Se existissem três ou mais *triggers* você poderia atribuir as suas posições os valores 10, 20 e 30. É uma boa ideia deixar um espaço entre a numeração para que você possa facilmente inserir outras *triggers* com pontos de execução entre as já criadas.

```

CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
ACTIVE
AFTER UPDATE
POSITION 0
AS
BEGIN
IF (old.salary <> new.salary) THEN
INSERT INTO salary_history
(emp_no, change_date, updater_id, old_salary, percent_change)
VALUES (
old.emp_no,
'NOW',
user,
old.salary,
(new.salary - old.salary) * 100 / old.salary);
END^

```

Após a palavra chave AS vem a declaração de qualquer variável local usando o comando DECLARE VARIABLE igual ao que foi usado para *stored procedures*. Finalmente vem o corpo da *procedure* delimitado pelos comandos BEGIN-END.

Uma coisa para se ter em mente quando estiver usando *triggers* é que uma simples alteração em um banco de dados pode causar o disparo de várias *triggers*. Uma alteração na tabela A pode disparar uma *trigger* que atualiza a tabela B. A atualização da tabela B, por sua vez, pode disparar uma *trigger* que insere um novo registro na tabela C o que pode provocar o disparo de uma *trigger* que atualiza a tabela D e assim sucessivamente.

O segundo ponto importante sobre *triggers* é que uma *trigger* é parte da transação que a disparou. Isto significa que se você inicia uma transação e atualiza uma linha que dispara uma *trigger* e esta *trigger* atualiza outra tabela que dispara outra *trigger* que atualiza outra tabela e você então dá um ROLLBACK na transação, tanto sua alteração quanto todas as alterações que foram feitas pela série de disparos de *triggers*, serão canceladas.

### **BEFORE ou AFTER?**

Uma *trigger* tem que ser disparada antes do registro ser atualizado caso você queira alterar o valor de uma ou mais colunas antes que a linha seja atualizada ou caso você queira bloquear a alteração da linha gerando uma *EXCEPTION*. Por exemplo, você teria de usar uma *trigger* BEFORE DELETE para evitar que o usuário deletasse o registro de um cliente que tenha comprado nos últimos dois anos..

*Triggers* do tipo AFTER são usadas quando você quer garantir que a atualização que disparou a *trigger* esteja completa com sucesso antes de você executar outras ações. A *trigger* acima é um bom exemplo. Esta *trigger* insere uma linha na tabela "salary\_history" sempre que o salário de um funcionário é alterado. A linha de histórico contém o salário antigo e o percentual de alteração. Como a atualização do registro do funcionário pode falhar por várias razões, como um valor em um campo que viola restrições impostas por exemplo, você não vai querer criar o registro de histórico até que a atualização seja completa com sucesso.

### **Usando OLD e NEW**

No exemplo de *trigger* acima você pode ver nomes de campos precedidos das palavras "OLD" e "NEW". No corpo de uma *trigger* o Firebird deixa disponíveis tanto o valor antigo como o novo valor de qualquer coluna, por exemplo old.salary e new.salary. Usando os valores OLD e NEW você pode facilmente criar registros de histórico, calcular o percentual de alteração de um valor numérico, encontrar em outras tabelas registros que combinem com

o valor antigo ou novo de um campo ou fazer qualquer outra coisa que você precise fazer.

## Gerando EXCEPTIONS

Em uma *trigger* do tipo BEFORE você pode evitar que a alteração que disparou a *trigger* seja efetivada, gerando uma EXCEPTION. Antes que você possa gerar uma EXCEPTION você precisa criá-la usando o comando CREATE EXCEPTION. Por exemplo:

```
CREATE EXCEPTION CUSTOMER_STILL_CURRENT
'Este cliente comprou nos últimos dois anos.'
```

Onde as palavras-chave CREATE EXCEPTION são seguidas do nome da exceção e do texto da mensagem de erro para esta exceção. Para gerar esta EXCEPTION em uma *trigger* ou *stored procedure* use a palavra chave EXCEPTION como mostrado abaixo:

```
EXCEPTION CUSTOMER_STILL_CURRENT;
```

Quando você gera uma EXCEPTION a execução da *trigger* ou da *stored procedure* é terminada. Qualquer comando na *trigger* ou *stored procedure* depois da exceção não será executado. No caso de uma *trigger* do tipo BEFORE a atualização que disparou a *trigger* é abortada. Finalmente a mensagem de erro da exceção é retornada para a aplicação. Você pode deletar uma EXCEPTION usando o comando DROP EXCEPTION e alterar a mensagem associada usando o comando ALTER EXCEPTION. Por exemplo:

```
ALTER EXCEPTION CUSTOMER_STILL_CURRENT 'Este cliente ainda está ativo.';
DROP EXCEPTION CUSTOMER_STILL_CURRENT;
```

## Usando GENERATORS

O Firebird não tem um tipo de campo autoincrementável. Ao invés disso tem uma ferramenta mais flexível chamada GENERATOR. Um GENERATOR retorna um valor incrementado toda vez que você o chama. Para criar um GENERATOR use o comando CREATE GENERATOR como a seguir.

```
CREATE GENERATOR CUSTOMER_ID;
```

Para excluir um GENERATOR basta usar o comando DROP GENERATOR, note que esse comando só é válido para o Firebird já que no Interbase isso não é possível, pelo menos até a sua versão 6. Para obter o próximo valor de um GENERATOR use a função GEN\_ID() por exemplo:

```
GEN_ID(CUSTOMER_ID, 1);
```

O primeiro parâmetro é o nome do GENERATOR e o segundo é o incremento. No exemplo o valor retornado será o último valor mais um. O Incremento pode ser qualquer valor inclusive zero, o que é muito útil para se obter o valor corrente de um GENERATOR sem alterar seu valor. Você pode também alterar o valor de um GENERATOR a qualquer momento usando o comando SET GENERATOR com a seguir:

```
SET GENERATOR CUSTOMER_ID TO 1000;
```

Note que se você chamar GEN\_ID dentro de uma transação e então executar um ROLLBACK o valor do GENERATOR não retornará ao valor anterior. Ele não é influenciado pelo ROLLBACK. GENERATORS são frequentemente usados em *triggers* para fornecer um valor para uma chave primária como no exemplo a seguir:

```
CREATE TRIGGER SET_EMP_NO FOR EMPLOYEE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    new.emp_no=gen_id(emp_no_gen, 1);
END ^
```

Você pode também chamar GEN\_ID em uma *stored procedure* que retorna o valor do GENERATOR em um parâmetro de saída para a aplicação cliente. O cliente pode então atribuir o valor a sua chave primária e apresentá-la ao usuário quando este cria um novo registro antes mesmo que este tenha sido gravado com um POST. Neste caso você poderá querer também criar uma *trigger* como a mostrada acima para o caso de um cliente inserir um registro e não fornecer um valor para a chave primária. Tudo que você tem de fazer é alterar o código como a seguir:

```
IF new.emp_no IS NULL THEN new.emp_no = gen_id(emp_no_gen, 1);
```

Agora a *trigger* só ira fornecer um valor para a chave primária no caso do campo ser nulo.

### **Alterando e excluindo Triggers**

Você pode também usar o comando ALTER TRIGGER para alterar tanto o cabeçalho quanto o corpo da *trigger*. O uso mais comum da alteração do cabeçalho é ativar e desativar uma *trigger*. Outro uso é trocar o POSITION da *trigger*. O comando a seguir irá inativar a *trigger*.

```
ALTER TRIGGER SET_EMP_NO INACTIVE;
```

Para alterar apenas o corpo da *trigger* forneça seu nome sem as outras informações do cabeçalho e então o novo corpo como mostrado a seguir. Você pode também alterar o cabeçalho e o corpo ao mesmo tempo.

```
ALTER TRIGGER SET_EMP_NO
AS
BEGIN
IF new.emp_no IS NULL THEN new.emp_no = gen_id(emp_no_gen, 1);
END ^
```

Para apagar uma *trigger* use o comando DROP TRIGGER. Por exemplo:

```
DROP TRIGGER SET_EMP_NO;
```

## **Sumário**

*Stored procedures* e *triggers* são o coração do desenvolvimento de aplicações cliente/servidor. Usando *stored procedures* e *triggers* você pode:

1. Reduzir o tráfego de rede.
2. Criar um conjunto comum de regras de negócio no banco de dados que se aplicará a todas as aplicações cliente.
3. Fornecer rotinas comuns que estarão disponíveis para todas as aplicações cliente reduzindo assim o tempo de desenvolvimento e manutenção.
4. Centralizar o processamento no servidor e reduzir os requisitos de hardware nas estações cliente.
5. Aumentar a performance das aplicações.

Para mais informações sobre o uso de *stored procedures* e *triggers* consulte “Data Definition Guide” e “Language Guide” como também as *stored procedures* e *triggers* no banco de dados de exemplo EMPLOYEE.GDB.

<p>Artigo Original:</p> <p><b>Bill Todd</b></p> <p>Borland Developers Conference San Diego 2000</p>	
<p>Tradução e adaptação:</p> <p><b>Alessandro Fernandes</b></p> <p><a href="mailto:alc.fernandes@uol.com.br">alc.fernandes@uol.com.br</a></p>	<p>Comunidade Firebird de Língua Portuguesa</p> <p>Visite a Comunidade em:</p> <p><a href="http://www.comunidade-firebird.org">http://www.comunidade-firebird.org</a></p>
<p>A Comunidade Firebird de Língua Portuguesa foi autorizada pelo Autor do Original para elaborar esta tradução.</p>	