



FACULDADES INTEGRADAS CÂNDIDO RONDON

Curso Superior de Tecnologia em Processamento de Dados

Banco de Dados

Prof.: Cristiano Maciel

FIREBIRD – UM SGBD ROBUSTO E GRATUITO

REVISÃO/EDITORAÇÃO/ARTE FINAL:

Elton Ricelli Ferreira de Rezende (elton.npd@unirondon.br) e

Rafael Nantes Flores (rafael.npd@unirondon.br)

Cuiabá-MT, outubro de 2003



SUMÁRIO

1. FIREBIRD E SEUS CONCEITOS	5
1.1. VOCÊ CONHECE O FIREBIRD?	5
1.2. NOMENCLATURA DOS TIPOS DE DADOS	8
1.3. PRINCIPAIS FERRAMENTAS DE MANUTENÇÃO DO FIREBIRD (WINDOWS)	10
2. SQL – STRUCTURED QUERY LANGUAGE	13
2.1. COMANDOS E FUNÇÕES :	13
2.2. COMANDOS DE ALTERAÇÃO:	14
2.2.1. ALTER DATABASE	14
2.2.2. ALTER DOMAIN	14
2.2.3. ALTER EXCEPTION	14
2.2.4. ALTER INDEX	14
2.2.5. ALTER PROCEDURE	14
2.2.6. ALTER TABLE	15
2.2.7. ALTER TRIGGER	15
2.3. FUNÇÕES INCLUSAS NO FIREBIRD	15
2.3.1. AVG()	15
2.3.2. CAST()	15
2.3.3. COMMIT	15
2.3.4. COUNT()	15
2.4. COMANDOS DE CRIAÇÃO	15
2.4.1. CREATE DATABASE	15
2.4.2. CREATE DOMAIN	16
2.4.3. CREATE EXCEPTION	16
2.4.4. CREATE GENERATOR	16
2.4.5. CREATE INDEX	16
2.4.6. CREATE PROCEDURE	17
2.4.7. CREATE TABLE	17
2.4.8. CREATE TRIGGER	18
2.4.9. CREATE VIEW	19
2.5. DECLARAÇÃO DE FUNÇÃO EXTERNA	19
2.5.1. DECLARE EXTERNAL FUNCTION	19
2.6. COMANDOS DE DELEÇÃO	19
2.6.1. DELETE	19
2.6.2. DROP DATABASE	19
2.6.3. DROP DOMAIN	19
2.6.4. DROP EXCEPTION	20
2.6.5. DROP EXTERNAL FUNCTION	20
2.6.6. DROP INDEX	20
2.6.7. DROP PROCEDURE	20
2.6.8. DROP TABLE	20
2.6.9. DROP TRIGGER	20
2.6.10. DROP VIEW	20
2.7. OUTROS COMANDOS	20
2.7.1. EXECUTE PROCEDURE	20
2.7.2. GEN_ID()	21
2.7.3. INSERT	21
2.7.4. MAX()	21
2.7.5. MIN()	21
2.7.6. ROLLBACK	21
2.7.7. SELECT	21
2.7.8. UPDATE	23
2.7.9. UPPER()	23



3.	TIPOS DE DADOS NO FIREBIRD	24
3.1.	BLOB	24
3.2.	CHAR(N).....	25
3.3.	VARCHAR(N).....	25
3.4.	DATE	25
3.5.	TIME	25
3.6.	TIMEStamp.....	26
3.7.	DECIMAL	26
3.8.	NUMERIC	26
3.9.	SMALLINT.....	26
3.10.	INTEGER.....	26
3.11.	FLOAT.....	27
3.12.	DOUBLE PRECISION	27
4.	STORED PROCEDURES E TRIGGERS NO FIREBIRD.....	28
4.1.	INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 01)	28
4.2.	INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 02)	30
4.3.	INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 03)	32
4.4.	INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 04)	37
4.5.	TRIGGERS	39
5.	ANEXOS (DICAS).....	43
5.1.1.	EXCLUIR CÓDIGO-FONTE DE STORED PROCEDURE.....	43
5.1.2.	OBTER OS CAMPOS DA CHAVE-PRIMÁRIA	43
5.1.3.	OBTER A DATA DO SERVIDOR	43
5.1.4.	LISTAR AS TABELAS E VIEWS DO BANCO DE DADOS.....	43
5.1.5.	BACKUP E RESTORE COM GBAK.....	43
5.1.6.	CRIAR E USAR DOMÍNIOS (DOMAIN'S).....	44
6.	REFERÊNCIAS BIBLIOGRÁFICAS	45



ÍNDICE DE FIGURAS

FIGURA 01: PLATAFORMAS SUPORTADAS PELO FIREBIRD	8
FIGURA 02: LICENÇAS	9
FIGURA 03: USUÁRIOS	9
FIGURA 04: COMPARATIVO	10
FIGURA 05: IBEXPERT – REGISTRANDO	10
FIGURA 06: IBEXPERT – PROPRIEDADES	11
FIGURA 07: IBEXPERT – SCRIPT EXECUTE	11



1. FIREBIRD E SEUS CONCEITOS

Este capítulo foi extraído do artigo de Rodrigo Cardoso¹ [FIRE 01] e [FIRE 02], para fins de pesquisa e desenvolvimento nas nossas aulas de Banco de Dados.

1.1. VOCÊ CONHECE O FIREBIRD?

Se você procura um banco de dados fácil de utilizar, compacto, com recursos de um SGDB bom, grátis e principalmente confiável, precisa conhecer o Firebird. O Firebird é um banco de dados Cliente/Servidor relacional compatível com SQL-ANSI-92, foi desenvolvido a partir do Código do Interbase 6 para ser independente de plataformas e de sistemas operacionais. Uma de suas vantagens é a dispensa do uso de Administradores de Banco de Dados (DBA). Simples de ser utilizado, basta instalar o software, sem a interferência freqüente de profissionais na manutenção do banco, além disso, ele dispensa o uso de super-servidores, utilizando pouco espaço em disco e pouca memória em situações normais. Por isso a plataforma necessária para a sua instalação e utilização pode ser reduzida, diminuindo consideravelmente os custos do projeto. Seu desenvolvimento iniciou em meados de 1985 por uma equipe de engenheiros da DEC (Digital Equipment Corporation). Tendo como nome inicial de Gton, o produto sofreu varias alterações até, finalmente em 1986 receber o nome de Interbase® iniciando na versão 2.0 e hoje esta na versão 7. Desenvolvido por um grupo independente de programadores voluntários, su código-fonte é baseado no InterBase(tm), disponibilizado pela Borland ao abrigo da "InterBase Public License v.1.0" em 25 de Julho de 2000. Esta versão foi compilada a partir de código que foi sujeito a uma extensa limpeza, eliminação de "bugs" e várias versões "beta", durante o ano que passou entre a disponibilidade do código fonte e esta versão. A "On-Disk Structure" é ainda a ODS 10. Algumas funcionalidades que obrigam a alterar a ODS foram adiadas para uma versão posterior. Foram adicionadas algumas extensões à linguagem, mas hoje, ainda são compatíveis as linguagens, por exemplo, se você tem um sistema rodando Interbase 6, pode migrar para o Firebird sem problemas, apenas removendo o Interbase e instalando o Firebird, recomenda-se que se faça um backup e um restore no banco após a migração. O publico alvo do Firebird são pequenas, médias e grandes empresas que procuram um banco de dados confiável, pratico e grátis. Uma pesquisa realizada pelo site Firebase(www.firebase.com.br), mostra que 50,37% dos usuários do site utilizam Firebird e os outros 49,63% utilizam versões do Interbase (5,6,7), isso mostra que mesmo sendo um banco "novo" no mercado, o Firebird esta crescendo muito. Sua utilização é bastante vasta, podendo ser usado tanto em aplicações Client/Server como em páginas da Internet, existem vários sites utilizando e aprovando seu desempenho.

Se o Firebird é tão bom, porque ele não é tão badalado como o Oracle, o Microsoft SQL server e outros servidores SQL? O Firebird não é tão famoso quanto o Oracle, ou o SQL Server porque ainda é pouco conhecido, muita gente não sabe de suas qualidades, mas, isso tende a mudar gradativamente, hoje em dia, no Brasil por exemplo, já temos muito material na Internet sobre Firebird, o que a pouco tempo atrás não existia. Segue algumas de suas especificações:

¹ Rodrigo Cardoso é Analista de Sistemas da DataSystems Informática.



OPEN SOURCE

Significa que possui seu código fonte disponibilizado. Você pode baixar na Internet a instalação, os fontes e sua documentação, instalá-lo e utilizá-lo normalmente, sem custo algum.

SUORTE A PROTOCOLOS DE REDE

Protocolo é um conjunto de regras que definem como os dados serão transmitidos; como será feito o controle de erros e retransmissão de dados; como os computadores serão endereçados dentro da rede etc. Um micro com o protocolo NetBEUI instalado, só será capaz de se comunicar através da rede com outros micros que também tenham o protocolo NetBEUI, por exemplo. É possível que um mesmo micro tenha instalados vários protocolos diferentes, tornando-se assim um “poliglota”. O Firebird suporta os seguintes protocolos:

- TCP/IP para todas as plataformas;
- NetBEUI;
- IPX/SPX;

COMPATIBILIDADE COM ANSI SQL-92

Isso facilita e muito quem esta migrando de outros bancos para o Firebird, pois o ANSI SQL é o padrão internacional utilizado pela grande maioria de sistemas de bases de dados.

INSTALAÇÃO RÁPIDA

Você faz a instalação bem rápido e sem dificuldades. O firebird ocupa pouco espaço em disco, para efetuar o download acesse o site: www.ibphoenix.com.

STORED PROCEDURES (PROCEDIMENTOS ARMAZENADOS)

São procedimentos armazenados no banco que executam várias tarefas, o Firebird possui um tipo de linguagem de programação com comandos FOR, IF, etc. As Stored Procedures possuem a grande vantagem de serem executadas diretamente no Servidor, fazendo com que a rede fique menos carregada, a regra para utilizá-las é bem simples, onde der pra usar Stored Procedures use, testes comprovam que o uso delas aumenta e muito a velocidade da aplicação. Abaixo segue 5 motivos pra utilizá-las:

- => Reduzir o Tráfico de Rede.
- => Criar um conjunto comum de regras de negócio no banco de dados que se aplicará a todas as aplicações cliente.
- => Fornecer rotinas comuns que estarão disponíveis para todas as aplicações cliente reduzindo assim o tempo de desenvolvimento e manutenção.
- => Centralizar o processamento no servidor e reduzir os requisitos de hardware nas estações cliente.
- => Aumentar a performance das aplicações.

TRIGGERS (GATILHOS)

Triggers são bem parecidas com Stored Procedures, exceto que triggers são executadas automaticamente quando uma alteração ocorre a tabela na qual ela esta conectada, além disso, triggers não possuem parâmetros de entrada e não retornam valores.



ACESSO SIMULTÂNEO A MÚLTIPLOS BANCOS DE DADOS

Uma ou mais aplicações podem acessar vários bancos de dados ao mesmo tempo. Por exemplo, você pode ter um banco de dados onde é guardado informações do seu sistema interno e um de sua página da Internet. Sua aplicação pode acessar tanto os dados internos como da página da Internet.

CAMPOS BLOB

Tipo de dados cujo tamanho é aumentado dinamicamente, podendo conter texto ou dados não formatados, como fotos, textos, resumindo qualquer tipo de arquivo.

OTIMIZAÇÃO DE QUERIES

Queries são comandos que serão executados pelo banco de dados, esses comandos podem ser tanto consultas, como alterações. O Firebird otimiza queries(consultas,atualizações) automaticamente, ou você pode especificar um plano(PLAN) para a queries, facilitando e muito.

FUNÇÕES DE USUÁRIO (UDFs)

Módulos de programa que rodam no servidor, adicionando funções ao servidor de acordo com as necessidades do usuário. Uma função definida pelo usuário (UDF) é meramente uma função escrita em qualquer linguagem de programação capaz de gerar uma biblioteca compartilhada. No Windows, essas bibliotecas são mais conhecidas pelo nome de Dynamic Link Libraries (DLL's).

JUNÇÕES EXTERNAS (OUTER JOINS)

Construção de relações entre duas tabelas que habilita operações complexas.

MATRIZES MULTIDIMENSIONAIS

Suporte nativo a arrays multidimensionais utilizados em aplicações científicas e financeiras. Um único campo do banco pode guardar um array de até 16 dimensões, simplificando o desenvolvimento e aumentando o desempenho destas aplicações.

SHADOWS

É um recurso de sombreamento de arquivos de banco de dados para prevenção de "crash" em disco rígido, falha de rede ou deleção do banco. A maior vantagem de se trabalhar com shadows está ligada à segurança. Problemas de falha em disco rígido ou remoção indevida de arquivos GDB são facilmente contornados, quando você tem uma "shadow" ela é uma cópia idêntica do seu arquivo de banco de dados que pode rapidamente ser ativada e tomar o papel do banco. No momento da criação das shadows você pode estar com usuários conectados, ou seja, neste momento não é necessário o uso exclusivo do banco. Shadows não requerem nenhuma manutenção ou administração.

TRAVAMENTO OTIMISTA

Quando um usuário processa uma alteração o registro não impedirá que outras pessoas tentem também alterar este registro. Quando um usuário começa a alterar um registro no Firebird, uma cópia do registro original é salva. O usuário executa seu serviço, mas os outros usuários não estão sob nenhuma forma impedidos de acessar o mesmo registro, esse tipo de travamento é chamado de Travamento Otimista. O Firebird utiliza tecnologia de travamento otimista para proporcionar grande taxa de uso de operações de banco de dados para clientes, ele implementa



travamentos a nível de linha reais para restringir mudanças somente nos registros do banco de dados que um cliente modifica, diferente de travamentos a nível de página, que restringe qualquer dado arbitrário que estiver armazenado fisicamente próximo no banco de dados. Travamentos a nível de linha permitem múltiplos clientes atualizarem dados em uma mesma tabela sem conflito, resultando em menor serialização das operações de bancos de dados.

API DO FIREBIRD

São funções do próprio Firebird que habilitam construção e acesso direto ao Firebird com o recebimento e retorno. Por exemplo você pode usar a API pra criar seu banco automaticamente, sem ter que enviá-lo junto com a aplicação para o Cliente.

MULTIPLATAFORMA

O Firebird trabalha em vários sistemas operacionais, veja a tabela abaixo.

Comparação entre Firebird e versões do Interbase				
	IB 5.5	IB 5.6	IB 6	Firebird
WINDOWS NT	Não recomendado	SIM	SIM	SIM
LINUX	NÃO	SIM	SIM	SIM
NETWARE	NÃO	SIM	NÃO	NÃO
SCO UNIX	SIM	NÃO	NÃO	EM TESTE
SOLARIS	SIM	NÃO	SIM	SIM
HP-UX	SIM	NÃO	NO	SIM
FreeBSD	NÃO	NÃO	NÃO	SENDO DES.
Mac-OS (Darwin)	NÃO	NÃO	NÃO	SIM

Figura 01: Plataformas suportadas pelo Firebird

SUPORTE A MÚLTIPLOS ARQUIVOS

Você pode dividir seu GDB em vários arquivos, no caso de arquivos muito grandes que não são suportados pelo S.O. O limite de tamanho do GDB é o limite do tamanho de um arquivo no Sistema Operacional.

USUÁRIOS DE PESO

Vejam quem está utilizando o Firebird: NASA, Motorola, Nokia, MCI, Northern Telecom, Philadelphia Stock Exchange, Bear Stearns, First National Bank of Chicago, Money Store, US Army, Boeing, IBM (Brasil), Intesis Tecnologia, Data System Informática.

1.2. NOMENCLATURA DOS TIPOS DE DADOS

O Firebird, usando Dialeto 3, suporta a maioria dos tipos de Dados do SQL, apenas não tem como tipo de dado, o tipo Boolean. Mas, isto não é uma falha, outro SGDB's também não tem este tipo de dado. Apesar de não ter este tipo de dado, podemos criar o nosso "tipo boolean" através de DOMAINS.

Os tipos de dados são BLOB, CHAR(n), VARCHAR(n), DATE, TIME, TIMESTAMP, DECIMAL, NUMERIC, SMALLINT, INTEGER, FLOAT, DOUBLE PRECISION.



Comparativo de preços (Estas informações foram retiradas da Borland Online Store).

Preços do Interbase 7:

Licença	IB 7 Novo	IB 7 Atualização	IB 6.x Novo	IB 6.x Atualização
Media kit*	\$50	\$30	\$50	\$30
Documentação impressa	\$25/manual	\$25/manual	Incl. w/ Media kit	Incl. w/ Media kit
Ativação do Servidor	\$200	\$130	\$200	\$130
CPU adicional	\$1000	n/a	n/a	n/a
1 licença de usuário	\$150	\$130	\$150	\$100
10 licenças de usuário	\$1200	\$780	\$1200	\$780
20 licenças de usuário	\$2100	\$1365	\$2100	\$1365
50 licenças de usuário	\$3500	\$2999	Não disponível	Não disponível
Internet Licença (usuários ilimitados)	\$2500	\$1625	\$2500	\$1625
Desktop Edition CD (2 licenças de usuário)	\$60	\$30	\$60	\$30
Desktop Edition 20 Usuários	\$800	\$?	\$800	\$?
Desktop Edition 100 Usuários	\$2000	\$1200	\$2000	\$1200

Figura 02: Licenças

Comparativo entre Interbase 7 e outros Bancos de Dados, lembrando que o Firebird é totalmente grátis:

Database Server	1 CPU, 50 users	1 CPU, 100 users	1 CPU, 150 users	2 CPUs, 100 users	2 CPUs, 150 users	4 CPUs, 200 users
InterBase 7	\$50 + \$200 + \$3500 = \$3750	\$50 + \$200 + \$7000 = \$7250	\$50 + \$200 + 10500 = \$10750	\$50 + \$200 + \$1000 + 7000 = \$8250	\$50 + \$200 + \$1000 + \$10500 = \$11750	\$50 + \$200 + \$3000 + \$14000 = \$17250
IBM DB/2 Workgroup Edition (WUE used when cheaper)	\$999 + (\$249 * 50) = \$13449	\$14250	\$14250	\$999 + (\$249*100) = \$25899	(\$14250 * 2) = \$28500	\$999 + (\$249 * 200) = \$50799
IBM DB/2 Enterprise Edition	\$20000	\$20000	\$20000	(\$20000 * 2) = \$40000	(\$20000 * 2) = \$40000	(\$20000 * 4) = \$80000
Microsoft SQL Server 2000 Standard Edition	\$4999	\$4999	\$4999	(\$4999 * 2) = \$9998	(\$4999 * 2) = \$9998	(\$4999 * 4) = \$19996
Microsoft SQL Server 2000 Enterprise Edition	\$19999	\$19999	\$19999	(\$19999 * 2) = \$39998	(\$19999 * 2) = \$39998	(\$19999 * 4) = \$79996
Oracle 9i Standard Edition	\$15000	\$15000	\$15000	(\$15000 * 2) = \$30000	(\$15000 * 2) = \$30000	(\$15000 * 4) = \$60000
Oracle 9i Enterprise Edition	\$40000	\$40000	\$40000	(\$15000 * 2) = \$80000	(\$40000 * 2) = \$80000	(\$40000 * 4) = \$80000

Figura 03: Usuários



Comparação técnica

Comparativo	PostgreSQL	Oracle	MS SQL	MySQL	Interbase Free	Firebird	Interbase Comercial
* Open Source	X			X	X	X	
Plataformas							
* Linux	X	X		X	X	X	X
* FreeBSD	X			X			
* Windows	X	X	X	X	X	X	X
* Sun Solaris	X	X				X	
* Mac OS X	X					X	
* IBM AIX	X	X				X	
* HP UX	X	X				X	
Características							
* ACID	X	X					
* Stored Procedures / Triggers	X	X	X		X	X	X
* Transações Concorrentes	X	X					
* ANSI SQL 99	X						
* ANSI SQL 92	X	X	X		X	X	X
* Sem limites de usuários	X			X	X	X	
* Integridade Referencial	X	X	X		X	X	X
* Transações	X	X	X		X	X	X
* ODBC Free	X	X	X	X			

Figura 04: Comparativo

1.3. PRINCIPAIS FERRAMENTAS DE MANUTENÇÃO DO FIREBIRD (WINDOWS)

O Firebird possui várias ferramentas de manutenção, IBConsole(Interbase), IBAcces, Quick Desk, IBExpert entre outros. Eu particularmente utilizo o IBExpert por ser bem fácil de usar e bem completo. Antes de instalar o IBExpert, baixe o Firebird no site da IBPhoenix (www.ibphoenix.com).

Instale-o normalmente e depois execute o Firebird (/Dretório de Instalação/Bin/IBServer.exe), isso fará com que o Firebird entre em execução. Para que sempre que iniciar o Windows o servidor seja executado, adicione o IBServer.exe no Menu Iniciar do Windows. Agora baixe o IBExpert Personal no site do IBExpert(www.ibexpert.com). Lembrando que o IBExpert Personal é uma versão FreeWare do IBExpert. Abra o IBExpert, vamos nos conectar ao Banco Employee.gdb que acompanha os exemplos do Firebird 1.0. Clique em Database depois em Register DataBase, como na figura abaixo:

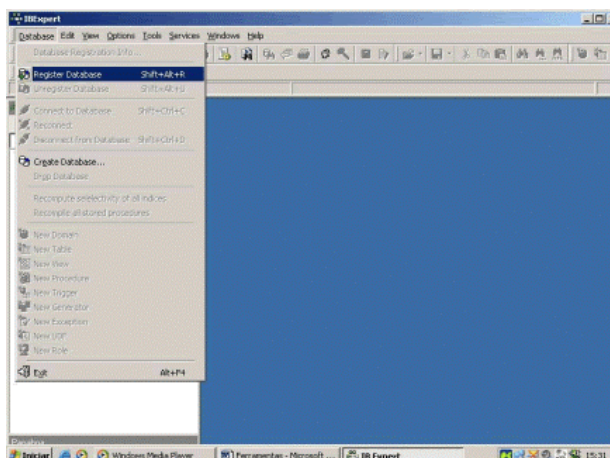


Figura 05: IBExpert – Registrando



Agora, preencha os dados do Formulário como na figura abaixo, em Server o valor LOCAL, em Database File, informe o diretório da sua instalação do Firebird mais “\examples\EMPLOYEE.GDB”.

Ex: c:\arquivos de programas\firebird\example\EMPLOYEE.GDB. No Database Alias informe Exemplo, no User Name informe SYSDBA e na Senha masterkey (minúsculo). SYSDBA é o usuário principal do Firebird, já pré definido e com acesso completo.

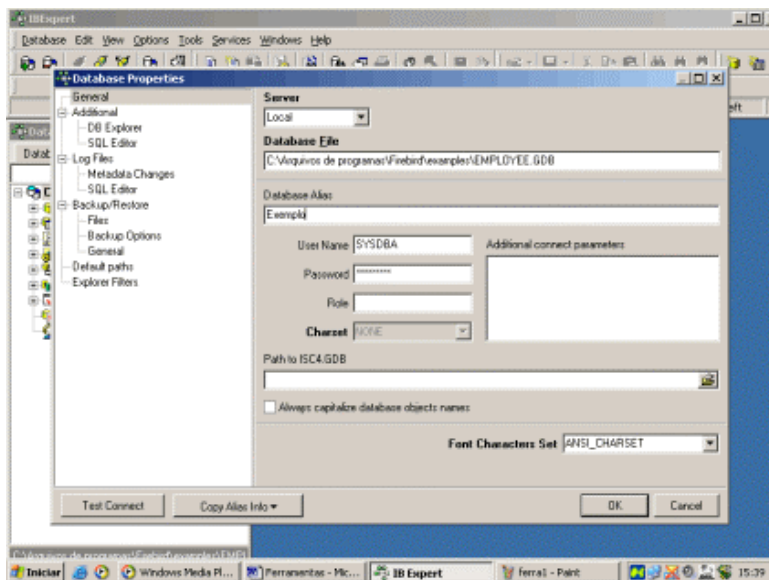


Figura 06: IBExpert – Propriedades

Do lado esquerdo do vídeo aparecerá agora o banco Exemplo, de duplo clique sobre ele para que ele se expanda e mostre os detalhes desse banco, clicando em tabelas, se expandirá as tabelas do banco EMPLOYEE. Mais um duplo click sobre a tabela COUNTRY, veremos suas informações, como: Fields, Constraints, Dependencies, etc.. Clicando na Aba Data, você tem os seus registros em um Grid, podendo alterá-los. Para executar comandos SQL no banco de dados, selecione no Menu Tools->Script Executive, ou Ctrl+F12, marque a opção Use Current Connection, como na figura abaixo. Agora é só digitar as linhas SQL e clicar em Run Script ou pressionar F9.

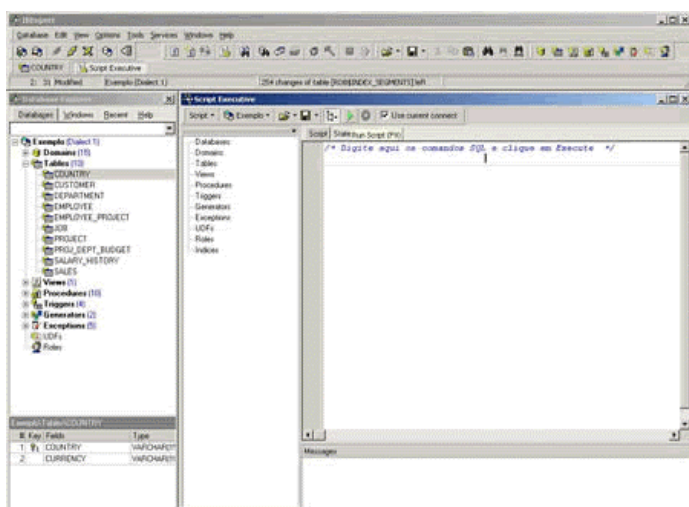


Figura 07: IBExpert – Script Execute



Vamos agora, criar uma tabela nova no nosso Banco EMPLOYEE, no Script Executive, digite o seguinte:

```
CREATE TABLE TABTESTE(  
  CODIGO INTEGER NOT NULL,  
  NOME VARCHAR(30) NOT NULL,  
  SALARIO NUMERIC(12,2),  
  PRIMARY KEY(CODIGO));
```

Clique em Run Script ou pressione F9, agora vá até as tabelas do nosso banco registrado e repare que já está criada a tabela TABTESTE, com os campos que definimos e com o campo Código como chave primária. Para alimentarmos a tabela TABTESTE, podemos dar duplo click nela e abrir a aba Dados, ou então rodarmos Scripts utilizando o comando INSERT INTO, por exemplo:

```
INSERT INTO TABTESTE (Codigo, Nome, Salario) VALUES(1, 'Rodrigo Cardoso', 1588);
```

CONCLUSÃO

O Firebird satisfaz as expectativas de quem precisa de um banco confiável, robusto, e Open Source. A algum tempo atrás poderíamos até reclamar do Firebird sobre pouca documentação, mas, hoje em dia temos muitas fontes e listas de discussão na Internet, e tudo isso faz com que o Firebird fique cada vez mais forte. Esta na hora de você rever seus conceitos sobre banco de dados. Vimos um exemplo simples de acesso ao Firebird, agora mãos a obra!



2. SQL – STRUCTURED QUERY LANGUAGE

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 03], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

Quando os Bancos de Dados Relacionais estavam sendo desenvolvidos, foram criadas várias linguagens destinadas à sua manipulação. O Departamento de Pesquisas da IBM desenvolveu a SQL como forma de interface para o sistema de Banco de Dados relacional denominado SYSTEM R, no início dos anos 70. Em 1986 o American National Standard Institute (ANSI), publicou um padrão SQL e ela se estabeleceu como linguagem padrão de Banco de Dados Relacional. A SQL apresenta uma série de comandos que permitem a definição dos dados, chamada de DDL (Data Definition Language - Definição de Dados Declarados), composta entre outros pelos comandos Create, que é destinado à criação do Banco de Dados, das tabelas que o compõe, além das relações existentes entre as tabelas. Como exemplo da classe DDL temos os comandos Create, Alter, Drop e Rename. Os comandos da série DML (Data Manipulation Language - Manipulação de Dados Declarados), destinados a consultas, inserções, exclusões e alterações em um ou mais registros de uma ou mais tabelas de maneira simultânea. Como exemplo de comandos da classe DML temos os comandos Select, Insert, UpDate, Delete, Commit e Rollback. Uma subclasse de comandos DML, é a DCL (Data Control Language - Controle de dados Declarados), que dispõe de comandos de controle como Grant, Revoke e Lock. A Linguagem SQL tem como grande virtude a sua capacidade de gerenciar índices, sem a necessidade de controle individualizado de índice corrente, algo muito comum nas linguagens de manipulação de dados do tipo registro a registro. Outra característica muito importante disponível em SQL é sua capacidade de construção de visões, que são formas de visualizarmos os dados, como listagens independentes das tabelas e organização lógica dados dados.

Outra característica interessante na linguagem SQL é a capacidade que dispomos de cancelar uma série de atualizações ou de as gravarmos, depois de iniciarmos uma seqüência de atualizações. Os comandos Commit e Rollback são responsáveis por estas facilidades. Devemos notar que a linguagem SQL consegue implementar estas soluções, somente pelo fato de estar baseada em Banco de Dados, que garantem por si mesmo a integridade das relações existentes entre as tabelas e seus índices.

2.1. COMANDOS E FUNÇÕES:

A seguir serão listados alguns comandos e funções mais utilizadas do Firebird, com parâmetros mais comuns. Não serão abordados todos os comandos, mas o essencial para se obter um bom conhecimento e conseguir usufruir do potencial desse banco de dados.

ALTER DATABASE	CREATE EXCEPTION	DROP PROCEDURE
ALTER DOMAIN	CREATE GENERATOR	DROP TABLE
ALTER EXCEPTION	CREATE INDEX	DROP TRIGGER
ALTER INDEX	CREATE PROCEDURE	DROP VIEW
ALTER PROCEDURE	CREATE TABLE	EXECUTE PROCEDURE
ALTER TABLE	CREATE TRIGGER	GEN ID()



ALTER TRIGGER	CREATE VIEW	INSERT
AVG()	DECLARE EXTERNAL FUNCTION	MAX() / MIN()
CAST()	DELETE	ROLLBACK
CLOSE	DROP DATABASE	SELECT
COMMIT	DROP DOMAIN	SET GENERATOR
COUNT()	DROP EXCEPTION	SUM()
CREATE DATABASE	DROP EXTERNAL FUNCTION	UPDATE
CREATE DOMAIN	DROP INDEX	UPPER()

2.2. COMANDOS DE ALTERAÇÃO:

Alteram banco de dados, domínios, exceções, etc.

2.2.1. ALTER DATABASE

Adiciona arquivos secundários ao Banco de Dados. Isso significa que poderemos ter um banco de Dados com vários arquivos dentro do mesmo GDB. Para a alteração da Base de Dados, o usuário SYSDBA, precisa ter acesso exclusivo ao Banco de Dados Firebird. Este comando, auxilia na repartição do Banco, deixando em algumas vezes o acesso aos Dados mais rápido.

Sintaxe :

ALTER [DATABASE | SCHEMA] ADD FILE 'nome' [LENGTH = PAGES | STARTING AT PAGE]

Ex : ALTER DATABASE

ADD FILE 'FATURAMENTO.GD1' STARTING AT PAGE 10001 LENGTH 10000 ADD FILE 'ESTOQUE.GD1' LENGTH 10000;

2.2.2. ALTER DOMAIN

Altera a definição de um domínio que já tenha sido criado. Pode-se alterar qualquer elemento de domínio, exceto os domínio de NOT NULL e a troca do tipo de Dado. Para redefinir o tipo de domínio e ou alterar o NOT NULL, deve apagar o domínio e criá-lo novamente. Atento para que se alguma tabela estiver usando o Domínio no qual você quer alterar os itens citados acima, você precisará deletar a coluna da tabela para ter sucesso no processo de troca. Aliás, o Firebird não deixará você trocar o tipo e ou a constraint NOT NULL, enquanto encontrar referências para este domínio. A criação de domínio, requer uma certa análise, para não encontrar este tipo de referência.

Sintaxe :

ALTER DOMAIN name { [SET DEFAULT { literal | NULL | USER}] [DROP DEFAULT] } [ADD [CONSTRAINT] CHECK (<dom_search_condition>)]

[[DROP CONSTRAINT] | new_col_name | TYPE data_type];

Ex : CREATE DOMAIN D_MES AS SMALLINT CHECK(VALUE BETWEEN 1 AND 12);

ALTER DOMAIN D_MES SET DEFAULT 1;

2.2.3. ALTER EXCEPTION

Altera a mensagem associada a uma exceção.

Sintaxe :

ALTER EXCEPTION nome_da_excecao 'Novo Texto';

Ex : CREATE EXCEPTION E_VALOR_INVALIDO 'Valor Inválido';

ALTER EXCEPTION E_VALOR_INVALIDO 'O Valor informado não é válido';

2.2.4. ALTER INDEX

Torna um índice ativo e ou inativo. Este comando está relacionado diretamente na performance do índice no seu Banco de Dados. Em certos momentos, o índice no Firebird pode ficar desbalanceado, desta forma, este comando recria o índice do Firebird.

Sintaxe :

ALTER INDEX name {ACTIVE | INACTIVE};

Ex : CREATE INDEX INDEX_NOME_TABELAX ON CLIENTES(NOME);

ALTER INDEX INDEX_NOME_TABELAX INACTIVE;

ALTER INDEX INDEX_NOME_TABELAX ACTIVE;

2.2.5. ALTER PROCEDURE

Altera uma Stored Procedure existente no Firebird. A sintaxe é a mesma da Create Procedure, se muda apenas o CREATE por ALTER. A sintaxe ALTER PROCEDURE, só não pode ser usada para alterar o nome da Stored Procedure. Mas, todos os seus itens, Parâmetros e o Corpo da Procedure pode ser alterada.

Sintaxe :

ALTER PROCEDURE name[(var datatype [, var datatype ...])]

[RETURNS (var datatype [, var datatype ...])]

AS

Begin

//Linguagem da Procedure.

end;

Ex : ALTER PROCEDURE SOMA_VENDAS_NO_MES (PMES SMALLINT) RETURNS (SOMA NUMERIC(18,02)) AS

BEGIN



```
SELECT SUM(VALOR_VENDA) FROM VENDAS WHERE MES :PMES INTO SOMA  
END
```

2.2.6. ALTER TABLE

Altera a estrutura de uma tabela e ou a integridade da mesma.
Sintaxe :

```
ALTER TABLE table ADD <col_def> <col_def> = col { <datatype> | [COMPUTED [BY] (<expr>) | domain][DEFAULT { literal |  
NULL | USER}][NOT NULL] [ <col_constraint>]  
[COLLATE collation]<col_constraint> = [CONSTRAINT constraint] <constraint_def>  
[ <col_constraint>]  
Ex : ALTER TABLE FORNECEDORES ADD CGC CHAR(14), DROP TIPOFORNECEDOR, ADD CONSTRAINT E_MAIL  
CHECK ( E_MAIL CONTAINING '@' OR E_MAIL IS NULL )
```

2.2.7. ALTER TRIGGER

Altera a definição de uma Trigger. Caso algum argumento for omitido, é assumido o valor definido no CREATE TRIGGER, ou no ALTER TRIGGER anteriormente usado.

Sintaxe :
ALTER TRIGGER name
[ACTIVE | INACTIVE] [(BEFORE | AFTER) {DELETE | INSERT | UPDATE}]
[POSITION number]
AS < trigger_body>;
Ex : ALTER TRIGGER "BAIXA_ESTOQUE" INACTIVE;
ALTER TRIGGER "EXCLUI_ITENS" FOR "TABELAVENDAS" BEFORE DELETE AS
BEGIN
DELETE FROM ITENS: WHERE NUM_VENDA = OLD.NUM_VENDA
END

2.3. FUNÇÕES INCLUSAS NO FIREBIRD

Funções básicas contidas no padrão SQL Ansi 92.

2.3.1. AVG()

Retorna a média de valores de uma coluna.
Ex: SELECT MES, AVG(VALOR_DA_VENDA) FROM VENDAS ORDER BY MES

2.3.2. CAST()

Usado em colunas, onde há a necessidade de se converter tipos de dados para outro formato.
Ex : SELECT CODIGO FROM FORNECEDORES WHERE DATA >= CAST(: DATA AS DATE)
SELECT CAST(CODIGO AS CHAR(10)) || ' - ' || NOME FROM VENDEDORES.
Neste exemplo foi usado para fazer a formatação e a "soma" entre dois campos. As barras verticais, foram usadas para concatenação, isto é, no Firebird e concatenação de campos, é através das barras ||, e não com o sinal de "+" como em outros SGDB's.
Este são os "Type Casting" aceitos pelo Firebird :
Caracter em Numérica ou Data
Numérico em Caracter ou Data
Data em Caracter ou Numérico

2.3.3. COMMIT

Grava as alterações de uma transação permanente no Banco de Dados.
Sintaxe :
COMMIT

2.3.4. COUNT()

Retorna a quantidade de registros para uma condição em um SELECT
Sintaxe :
COUNT(* | ALL | valor | DISTINCT valor)
Ex : SELECT COUNT(*) FROM CLIENTES

2.4. COMANDOS DE CRIAÇÃO

Criam banco de dados, domínios, exceções, generators, etc.

2.4.1. CREATE DATABASE

Cria um novo Banco de Dados ".GDB". Nele pode especificar as suas características, como :
Nome do Arquivo;
Tamanho da página de dados (PAGE SIZE);
Sintaxe :



```
CREATE {DATABASE | SCHEMA} ' filespec'
[USER ' username' [PASSWORD ' password']]
[PAGE_SIZE [=] int]
[LENGTH [=] int [PAGE[S]]]
[DEFAULT CHARACTER SET charset]
[ <secondary_file>];
<secondary_file> =FILE 'filespec' [<fileinfo>][<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[ <fileinfo>]
```

Ex :

```
CREATE DATABASE 'C:\DB\TESTE.GDB' DEFAULT CHARACTER SET ISO8859_1 FILE 'C:\DB\TESTE.GD1' STARTING AT
PAGE 10001 LENGTH 10000 PAGES
```

2.4.2. CREATE DOMAIN

Cria uma definição de um “novo tipo de dado”. Onde pode ser feito também, checagem de valor, isto é, regras para o dado ser gravado neste “novo tipo de dado”. Quando falo em novo tipo de dado, não é possível no Firebird “sem alterar os fontes do mesmo”, criar um novo tipo de dado, mas, com DOMAINS, nós usamos dados já existentes, mas, especificando o tamanho e a regra a ser seguida.

Sintaxe :

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT { literal | NULL | USER}]
[NOT NULL] [CHECK ( <dom_search_condition>)]
[COLLATE collation];
Ex : CREATE DOMAIN STRINGNOME AS VARCHAR(45);
CREATE DOMAIN IDCODIGO AS INTEGER DEFAULT 1000 CHECK ( VALUE > 1000 ) NOT NULL
CREATE TABLE "FORNECEDOR" (ID INTEGER NOT NULL PRIMARY KEY,
NOME STRINGNOME, ---- Usamos o Domínio criado acima ----);
```

2.4.3. CREATE EXCEPTION

Cria uma mensagem de erro, armazenada no servidor, e só pode ser usado em Stored Procedure e ou Trigger

Sintaxe :

```
CREATE EXCEPTION "NOME_DA_EXCEPTION" 'MENSAGEM'
Ex: CREATE EXCEPTION "NOME_INVALIDO" 'O Valor informado para o campo, é inválido'
Este trecho de código, está contigo dentro de uma Trigger.
IF (NEW.NOME = "") THEN
EXCEPTION NOME_INVALIDO;
```

2.4.4. CREATE GENERATOR

Cria um Generator de número inteiros e seqüenciais. O Generator em conjunto com Trigger e ou Stored Procedure, é usado para simular o campo auto-incremento dos Bancos Desktop's. Serve também para evitar chaves duplicadas em campos numéricos.

O Valor inicial é Zero, mas, é atualizado toda vez que é chamado a função GEN_ID(). O Generator pode ser usado para incrementar ou decrementar valores. Para se saber o código atual do Generator você passa o valor Zero "0" para o Gen_ID().

Sintaxe :

```
CREATE GENERATOR "NOME_DO_GENERATOR"
Ex : Vamos simular neste exemplo, um auto numérico seqüencial da tabela de fornecedores da coluna ID.
CREATE GENERATOR COD_FORNECEDOR;
CREATE TRIGGER "COD_AUTO_FORNECEDOR" FOR "FORNECEDORES" BEFORE INSERT POSITION 0 AS
BEGIN
NEW.ID := Gen_ID("COD_FORNECEDOR",1);
END
```

Para excluir um GENERATOR, ele tem que ser excluído direto da tabela de sistema do Firebird "RDB\$GENERATORS".

```
Ex : DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = 'NOME_DO_SEU_GENERATOR';
```

2.4.5. CREATE INDEX

Cria um índice para uma ou mais colunas específicas da tabela. O índice está ligado diretamente a performance do seu banco de dados. O conceito de índices em ambientes Desktop's "xBase, Access, Paradox" é muito diferente do conceito de índices em ambiente Client/Server. Um índice em ambiente Client/Server "Firebird, Oracle, DB2", não tem a função de organizar a tabela, pois, você tem o mesmo efeito com ORDER BY. A função de um índice em ambiente Client/Server, é de performance em primeiro lugar, caso o índice seja um PK "Primary Key", tem a função de manter a integridade da tabela, caso o índice seja um FK "Foreign Key", tem a função de relacionamento e integridade da tabela, caso o índice seja UNIQUE, tem a função de não deixar valores iguais serem incluídos.

As cláusulas ASCENDING e DESCENDING, tem a função de organizar da maneira desejada o índice. O valor default é ASC

Sintaxe :

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX index ON table ( col [, col ...]);
Ex : CREATE INDEX IND_DATA_VENDA ON "VENDAS" ( DATA_VENDA );
CREATE DESC INDEX IND_SALARIOS ON "FUNCIONARIOS" ( SALARIO );
CREATE UNIQUE INDEX IND_COD_PRODUTO ON "PRODUTOS" ( ID );
```




2.4.6. CREATE PROCEDURE

Cria uma Stored Procedure "SP" e, define seus parâmetros de entrada e saída e o corpo da procedure. Uma SP é escrita em "linguagem" Firebird, e armazenada no próprio Banco de Dados. Stored Procedure's aceita todas as sentenças de manipulação de Dados e algumas extensões avançadas do Firebird, como :

IF..THEN..ELSE, WHILE...DO, FOR SELECT..DO, EXCEPTIONS ...

Existem dois tipos de SP :

Select Procedure : Utilizada na cláusula FROM do comando SELECT, como se fosse uma tabela e ou uma View. Este tipo de SP, deve obrigatoriamente retornar uma ou mais linhas de dados, caso contrário, ocorre um erro. Para retornar uma linha, use SUSPEND, para retornar uma ou mais linhas, use FOR SELECT.... DO e SUSPEND. Isto fará com que os parâmetros de retorno sejam preenchidos com a linha de retorno.

Executable Procedure : Utilizada no comando EXECUTE PROCEDURE, para realizar uma ou mais tarefas, mas, não retorna dados.

Sintaxe :

```
CREATE PROCEDURE name
[( param datatype [, param datatype ...])]
[RETURNS ( param datatype [, param datatype ...])]
AS
<procedure_body>;
<procedure_body>=[<variable_declaration_list>
<block>
< variable_declaration_list>=
DECLARE VARIABLE var datatype;
[DECLARE VARIABLE var datatype; ...]
<block> =
BEGIN
< compound_statement>
[< compound_statement>...]
END
< compound_statement>={<block> | statement;}
```

Ex :

```
CREATE PROCEDURE RESUMO_VENDAS ( VENDEDOR INTEGER) RETURNS ( VALOR_TOTAL DOUBLE PRECISION,
MEDIA DOUBLE PRECISION VALOR_MIN DOUBLE PRECISION, VALOR_MAX DOUBLE PRECISION ) AS
BEGIN
SELECT SUM(VALOR), AVG(VALOR), MIN(VALOR), MAX(VALOR) FROM VENDAS WHERE IDVENDEDOR = :VENDEDOR
INTO :VALOR_TOTAL, :MEDIA, :VALOR_MIN, :VALOR_MAX;
EXIT;
END
SELECT * FROM RESUMO_VENDAS 10;
"10, Neste caso é o código do vendedor".
```

2.4.7. CREATE TABLE

Cria uma nova tabela no seu banco de dados Firebird.

Sintaxe :

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec']
(<col_def> [, <col_def> | <tconstraint> ...]);
<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
[DEFAULT {literal | NULL | USER}]
[NOT NULL]
[<col_constraint>]
[COLLATE collation]
<datatype> =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[<array_dim>]
| (DATE | TIME | TIMESTAMP) [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])][<array_dim>] = [[x:]y [, [x:]y ...]]
<expr> = A valid SQL expression that results in a single value.
<col_constraint> = [CONSTRAINT constraint]
{ UNIQUE
| PRIMARY KEY
| REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>)}
<tconstraint> = [CONSTRAINT constraint]
{{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...]) REFERENCES other_table
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]}
```

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



```
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>))
<search_condition> = <val> <operator> {<val> | (<select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> {>= | <=}
| <val> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| (<search_condition>)
| NOT <search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>
<val> = { col [<array_dim>] | :variable
| <constant> | <expr> | <function>
| udf ([<val> [, <val> ...]])
| NULL | USER | RDB$DB_KEY | ? }
[COLLATE collation]
<constant> = num | 'string' | charsetname 'string'
<function> = COUNT (*) | [ALL] <val> | DISTINCT <val>
| SUM ([ALL] <val> | DISTINCT <val>)
| AVG ([ALL] <val> | DISTINCT <val>)
| MAX ([ALL] <val> | DISTINCT <val>)
| MIN ([ALL] <val> | DISTINCT <val>)
| CAST (<val> AS <datatype>)
| UPPER (<val>)
| GEN_ID (generator, <val>)
```

```
Ex : CREATE TABLE PRODUTOS (
ID INTEGER NOT NULL,
NOME VARCHAR(50) NOT NULL,
DATA DATE DEFAULT CURRENT_DATE NOT NULL,
PRECO DOUBLE PRECISION ( CHECK PRECO > 0),
ESTOQUE INTEGER ( CHECK ESTOQUE > 0),
VALOR COMPUTED BY ( PRECO * ESTOQUE ),
CONSTRAINT PK_PRODUTOS PRIMARY KEY(ID));
```

2.4.8. CREATE TRIGGER

Cria uma Trigger “Gatilho” para a tabela especificada. Trigger é um gatilho disparado após alguma ação ocorrida na tabela, isto é, podem existir Trigger de Insert, Update e Delete. As Trigger pode ser definidas como “Before-Antes” e “After-Depois”. Também pode ser definido um número onde indica qual a seqüência de tireis a ser seguida. A Trigger só é disparada pela ação na tabela, não podendo ser disparada pela aplicação. Dentro da Trigger há duas formas de se referenciar as colunas das tabelas :

OLD.Coluna e NEW.Coluna. Onde OLD. referencia o valor anterior da coluna e NEW. referencia o novo valor da coluna.

Sintaxe :

```
CREATE TRIGGER name FOR table
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION number]
AS <trigger_body> terminator
<trigger_body> = [<variable_declaration_list>] <block>
<variable_declaration_list> =
DECLARE VARIABLE variable <datatype>;
[DECLARE VARIABLE variable <datatype>; ...]
<block> =
BEGIN
<compound_statement>
[<compound_statement> ...]
END
<datatype> = SMALLINT
| INTEGER
| FLOAT
| DOUBLE PRECISION
| {DECIMAL | NUMERIC} [(precision [, scale])]
| {DATE | TIME | TIMESTAMP}
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
| {int} [CHARACTER SET charname]
```



| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]

<compound_statement> = {<block> | statement;}

Ex:

SET TERM !!;

CREATE TRIGGER "TRG_VERIFICA_ESTOQUE" FOR "ITENS_NOTA" BEFORE INSERT POSITION 0 AS

BEGIN

IF (NOT EXISTS (SELECT * FROM PRODUTOS WHERE ID = NEW.ID AND ESTOQUE >= NEW.QTDE_VENDIDA)) THEN

EXCEPTION ACABOU_ESTOQUE;

END !!

SET TERM ; !!

2.4.9. CREATE VIEW

Cria uma nova visão dos dados já existentes em uma tabela. Uma VIEW é uma tabela normal, pode ser fazer o mesmo que uma tabela normal. Um dos impedimentos referentes a VIEW é o ORDER BY.

VIEW são utilizadas para realizar tarefas como :

Restringir o acesso dos usuários;

Mostrar apenas as colunas "x";

Filtragem de dados já pré formatados;

E de certa forma, serve também como Data WareHousing;

A VIEW fica armazenada no Banco de Dados, mas, é armazenado apenas a definição da VIEW.

Existem dois tipos de VIEW's :

VIEW Read-Only : Quando o resultado não pode ser editado.

VIEW Update : Views que podem ser editadas.

Sintaxe :

CREATE VIEW name [(view_col [, view_col ...])]

AS <select> [WITH CHECK OPTION];

Ex : CREATE VIEW TELEFONES (NOME, TELEFONE) AS SELECT NOME, TELEFONE FROM CLIENTES;

2.5. DECLARAÇÃO DE FUNÇÃO EXTERNA

Possibilita incrementar ao banco uma funcionalidade através de DLL's.

2.5.1. DECLARE EXTERNAL FUNCTION

Declara uma função externa ao Banco de Dados Firebird. A função, mais conhecida como UDF. O princípio de criação de UDF, no Windows, é de construir a função em DLL "Delphi, C, C++, VB", e após isto, declarar a função no Firebird.

Sintaxe :

DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)]

[, datatype | CSTRING (int) ...]

RETURNS {datatype [BY VALUE] | CSTRING (int)} [FREE_IT]

ENTRY_POINT 'entryname'

MODULE_NAME 'modulename';

Ex : DECLARE EXTERNAL FUNCTION ABREVIAR_NOME CHAR(60) RETURNS CHAR(60) BY VALUE ENTRY_POINT

"MINHAUDF" MODULE_NAME "MINHADLL.DLL"

2.6. COMANDOS DE DELEÇÃO

Existem deleções com o Delete e o Drop.

2.6.1. DELETE

Apaga um ou mais registros de uma tabela Firebird. Se não for utilizado a cláusula WHERE, será apagado todos os registros da tabela.

Sintaxe :

DELETE [TRANSACTION transacão] FROM table

{[WHERE <search_condition>] | WHERE CURRENT OF cursor};

Ex : DELETE FROM VENDAS WHERE DATA_VENDA <= '1-JAN-1999';

2.6.2. DROP DATABASE

Apaga o Banco de Dados ".GDB" Firebird. O Banco de Dados, só pode ser deletado pelo seu criador "Owner" e ou pelo SYSDBA do Banco de Dados.

Sintaxe :

DROP DATABASE;

Ex

DROP DATABASE;

2.6.3. DROP DOMAIN

Deleta um domínio previamente criado no Firebird. Se o Domínio estiver em uso por alguma tabela, para solucionar este problema, o campo tem que ser excluído e após isto apagar o Domínio.

Sintaxe :

DROP DOMAIN "name"

Ex : DROP DOMAIN "STRINGNOME";

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



2.6.4. DROP EXCEPTION

Deleta uma exceção previamente criada no seu Banco de Dados Firebird. Se a exceção estiver sendo usada por alguma Stored Procedure e ou Trigger, a exclusão falhará. Desta forma, o usuário precisa retirar a EXCEPTION da SP e ou Trigger e após isto executar novamente a EXCEPTION.

Sintaxe :

DROP EXCEPTION "name"

Ex : DROP EXCEPTION "ACABOU_ESTOQUE";

2.6.5. DROP EXTERNAL FUNCTION

Deleta do Banco de Dados Firebird a declaração do uso da UDF. Este comando não exclui da DLL, mas, a torna inacessível ao Banco de Dados Firebird. Se alguma SP ou Trigger estiver usando a UDF, ocorrerá um erro na execução dos comandos.

Sintaxe :

DROP EXTERNAL FUNCTION "name"

Ex : DROP EXTERNAL FUNCTION "ABREVIAR_NOME";

2.6.6. DROP INDEX

Deleta o índice definido pelo usuário do Banco de Dados Firebird.

Sintaxe :

DROP INDEX "name"

Ex : DROP INDEX IND_NOME;

DROP PROCEDURE

Deleta uma SP previamente criada pelo usuário. As SP que estão sendo referenciadas em Trigger, VIEW, não poderão ser excluídas.

Sintaxe :

DROP PROCEDURE "name"

Ex : DROP PROCEDURE RESUMO_VENDAS;

2.6.7. DROP PROCEDURE

Deleta uma SP previamente criada pelo usuário. As SP que estão sendo referenciadas em Trigger, VIEW, não poderão ser excluídas.

Sintaxe :

DROP PROCEDURE "name"

Ex : DROP PROCEDURE RESUMO_VENDAS;

2.6.8. DROP TABLE

Apaga uma tabela do Banco de Dados, e também os índices referenciados e trigger's que a tabela faz referencia.

Sintaxe :

DROP TABLE "name";

Ex : DROP TABLE "FORNECEDORES";

2.6.9. DROP TRIGGER

Apaga uma Trigger do banco de dados.

Sintaxe :

DROP TRIGGER "name";

Ex : DROP TRIGGER "TRG_VERIFICA_ESTOQUE";

2.6.10. DROP VIEW

Deleta uma VIEW do Banco de Dados Firebird. Se a VIEW estiver sendo referenciada em outra VIEW, SP, Trigger, não poderá ser apagada. Apenas a definição da VIEW é excluída do Banco de Dados, os dados da VIEW permanecerão intactos na tabela original.

Sintaxe :

DROP VIEW "name";

Ex : DROP VIEW "TABELA_PRECOS";

2.7. OUTROS COMANDOS

Vários.

2.7.1. EXECUTE PROCEDURE

Executa uma Stored Procedure, não "Seletável-Select".

Sintaxe :

EXECUTE PROCEDURE [TRANSACTION transaction]

name [:param [[INDICATOR]:indicator]]

[, :param [[INDICATOR]:indicator] ...]

[RETURNING_VALUES :param [[INDICATOR]:indicator]

[, :param [[INDICATOR]:indicator] ...]];

Ex : EXECUTE PROCEDURE CUSTO_DEPTO 4 RETURNING_VALUES :SOMA;



2.7.2. GEN_ID()

Retorna o valor do GENERATOR, isto é, pode retornar o valor do GENERATOR e ou incrementar/decrementar. É informado do nome do Generator e o valor do retorno do GENERATOR.

Sintaxe :

GEN_ID("generator",ID);

Ex : SELECT GEN_ID("GEN_FORNECEDOR,0) FROM RDB\$DATABASE;

2.7.3. INSERT

Comando responsável para adicionar um mais registros na tabela de Banco de Dados Firebird. Os campos que forem omitidos recebem valores NULOS "NULL".

Sintaxe :

INSERT [TRANSACTION transaction] INTO <object> [(col [, col ...])]

{VALUES (<val> [, <val> ...]) | <select_expr>;

<object> = tablename | viewname

<val> = {variable | <constant> | <expr>

| <function> | udf ([<val> [, <val> ...]])

| NULL | USER | RDB\$DB_KEY | ?

} [COLLATE collation]

<constant> = num | 'string' | charsetname 'string'

<function> = CAST (<val> AS <datatype>)

| UPPER (<val>)

| GEN_ID (generator, <val>)

Ex : INSERT INTO CLIENTES (ID,NOME) VALUES (1,'Nome do Cliente');

INSERT INTO VENDAS_OLD SELECT * FROM VENDAS WHERE DATA_VENDA = CURRENT DATE;

2.7.4. MAX()

Função que agrega e retorna o valor máximo de uma coluna.

Sintaxe :

MAX([ALL <col> | DISTINCT <col>)

Ex : SELECT MAX(SALARIO) FROM FUNCIONARIOS;

2.7.5. MIN()

Função que agrega e retorna o valor mínimo de uma coluna.

Sintaxe :

MIN([ALL <col> | DISTINCT <col>)

Ex : SELECT MIN(SALARIO) FROM FUNCIONARIOS;

2.7.6. ROLLBACK

Desfaz as mudanças ocorridas até o exato momento no Banco de Dados Firebird, sem que o comando COMMIT tenha sido executado. Este comando e o Commit fecham a transação aberta pela aplicação e ou ferramenta de gerenciamento as tabelas.

Sintaxe :

ROLLBACK

Ex : ROLLBACK;

2.7.7. SELECT

Este é o comando responsável pela obtenção dos dados da tabela, view's e ou Stored Procedures.

Sintaxe :

SELECT [TRANSACTION transaction]

[DISTINCT | ALL]

{* | <val> [, <val> ...]}

[INTO :var [, :var ...]]

FROM <tableref> [, <tableref> ...]

[WHERE <search_condition>]

[GROUP BY col [COLLATE collation] [, col [COLLATE collation] ...]

[HAVING <search_condition>]

[UNION <select_expr> [ALL]]

[PLAN <plan_expr>]

[ORDER BY <order_list>]

[FOR UPDATE [OF col [, col ...]]];

<val> = {

col [<array_dim>] | :variable

| <constant> | <expr> | <function>

| udf ([<val> [, <val> ...]])

| NULL | USER | RDB\$DB_KEY | ?

} [COLLATE collation] [AS alias]

<array_dim> = [[x:]y [, [x:]y ...]]

<constant> = num | 'string' | charsetname 'string'

<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)

| SUM ([ALL] <val> | DISTINCT <val>)

| AVG ([ALL] <val> | DISTINCT <val>)

| MAX ([ALL] <val> | DISTINCT <val>)

| MIN ([ALL] <val> | DISTINCT <val>)

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



```

| CAST (<val> AS <datatype>)
| UPPER (<val>)
| GEN_ID (generator, <val>)
<tableref> = <joined_table> | table | view | procedure
[( <val> [, <val> ...])] [alias]
<joined_table> = <tableref> <join_type> JOIN <tableref>
ON <search_condition> | (<joined_table>)
<join_type> = [INNER] JOIN
| {LEFT | RIGHT | FULL } [OUTER] JOIN
<search_condition> = <val> <operator> {<val> | (<select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> {>= | <=}
| <val> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| (<search_condition>)
| NOT <search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>
<operator> = {= | < | > | <= | >= | != | <> | !=}
<plan_expr> =
[JOIN | [SORT] [MERGE]] ((<plan_item> | <plan_expr>)
| {<plan_item> | <plan_expr>} ...)
<plan_item> = {table | alias}
{NATURAL | INDEX (<index> [, <index> ...]) | ORDER <index>}
<order_list> =
{col | int} [COLLATE collation]
[ASC[ENDING] | DESC[ENDING]]
[, <order_list> ...]

```

Começaremos explicando o que acontece nos bastidores do SELECT quando você executa o comando de pesquisa simples.

Ex. : SELECT código,nome FROM funcionarios

Após está linha de comando, veja o que acontece com o "motor" do Firebird.

Abre a sessão para troca de dados;

Abre a tabela informada;

Posiciona o cursor de leitura no início da tabela;

Repete até o fim da tabela;

Lê as linhas de dados;

Envia as linhas de dados;

Posiciona na próxima linha de dados;

Testa o laço de repetição;

Fecha a tabela;

Fecha sessão de troca de dados.

Operador e funções ligados ao SELECT, isto é, antes da cláusula WHERE

Literal e String - No Firebird, você pode preencher colunas com valores Literais e ou String's para colunas virtuais e ou colunas fixas da tabela de dados. Mostraremos para este exemplo, o que seria o preenchimento de uma literal em uma coluna virtual ou coluna fixa por literal e ou String.

Ex. Literal : SELECT 1 FROM TabFuncionarios

SELECT 1 as ID TabFuncionarios

Ex. String : SELECT 'Firebird Brasil' FROM TabFuncionarios

SELECT 'Firebird Brasil' as Nome FROM TabFuncionarios

Expressão || - A expressão conhecida para concatenação no Firebird é ||. Desta forma, você pode usa-la para concatenar colunas virtuais e ou colunas fixas. O operador "+" para o Firebird, é usado para cálculos matemáticos, desta forma, se precisar concatenar colunas, use o ||.

DISTINCT - Prevê a exclusão de linhas semelhantes do Result Set.

INTO:var[,var,]] - Cláusula que transfere o valor de uma coluna para a variável indicada ao SELECT. Se o Result Set retornar mais de uma linha de dados, ocorrerá um erro, indicando que está clausula apenas poderá retornar uma linha de dados.

UDF([<val> [, <val> ...]]) - O Firebird permite você criar funções definidas pelo usuário. E desta forma, você pode chamar esta função em instruções **SELECT**

USER - Variável de ambiente do Firebird, indicando o ID do usuário de conexão com o Firebird.

Operadores que fazem parte da cláusula **WHERE**

BETWEEN - Este operador testa se o valor da coluna encontra-se no intervalo declarado.

IN - Verifica se valor está contido no Sub-Conjunto de dados na coluna declarada.

ALL - Verifica se uma valor é igual a todos os valores retornados em um SubQuery(*).

ANY e **SOME** - Verifica se um valor está contido em qualquer valor retornado num SubQuery(*).

EXISTS - Verifica se um valor existe e ou está presente em pelo menos uma linha no retorno do SubQuery(*). Esta clausula pode conter também **NOT EXISTS**.



SINGULAR - Opera com semelhança ao **EXISTS**, com a diferença de que o valor tem que existir exatamente em uma ocorrência do SubQuery(*)).

CONTAINING - Testa se o valor passado a coluna, contém em uma parte da string. Está cláusula é CASE-SENSITIVE

STARTING WITH - Testa se a coluna inicia exatamente como indicado pelo valor passado.

Cláusula **UNION**

Executa a união de uma mais tabelas com o mesmo nome de colunas.

SUM()

Função de Agregação que retorna a soma dos valores da coluna

Sintaxe :

SUM([ALL <val> | DISTINCT <val>)

Ex : SELECT SUM(V valor) FROM VENDAS;

2.7.8. UPDATE

Comando responsável pela atualização da tabela no Banco de Dados Firebird. Update trabalha de forma semelhante ao DELETE "é claro, com sua enorme diferença", se não passarmos a cláusula WHERE, toda a coluna da tabela será atualizada.

Sintaxe :

UPDATE [TRANSACTION transaction] {table | view}

SET col = <val> [, col = <val> ...]

[WHERE <search_condition> | WHERE CURRENT OF cursor];

Ex : UPDATE CLIENTE SET DATA_INCLUSAO = CURRENT DATE;

2.7.9. UPPER()

Função que retorna uma string com todos os caracteres em maiúsculo.

Sintaxe :

UPPER(<col>);

Ex : CREATE DOMAIN SEXO AS CHAR(01) (CHECK VALUE = UPPER(VALUE));

SELECT UPPER(NOME) FROM CLIENTES;



3. TIPOS DE DADOS NO FIREBIRD

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 04], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

O Firebird, suporta a maioria dos tipos de Dados do SQL, apenas não tem como tipo de dado, o tipo Boolean. Mas, isto não é uma falha do Firebird, outro SGDB's também não tem este tipo de dado. Apesar de não ter este tipo de dado, podemos criar o nosso "tipo boolean" através de DOMAINS.

3.1. BLOB

O tipo de Dado BLOB, tem o tamanho variável, isto é, não sabemos na hora da criação do campo BLOB qual será o seu tamanho realmente, mas, o limite do campo Blob que está na documentação do Firebird, é de 64k por segmento.

Este tipo de campo é o tipo indicado para armazenar Textos Grandes "Memos", Fotos, Gráficos, Ícones, isto é, aparentemente não tem um tipo de dado que não possa ser armazenado no Campo Blob. Campos Blob's não podem ser indexados.

Saber qual o sub-tipo correto utilizar é essencial para criar aplicativos que se utilizem dos campos BLOBs. Os BLOBs se apresentam em 3 versões :

Sub-tipo 0 - Armazena dados em formato binário – Fotos, etc.

Sub-tipo 1 - Armazena dados em formato texto – Memos.

Sub-tipos definidos pelo usuário.

Além dos 2 Sub-tipos pré-definidos, também existem os Sub-tipos definidos pelo usuário. Esses tipos são determinados com o uso de valores negativos logo após a palavra SUB_TYPE. O número utilizado é um inteiro determinado arbitrariamente pelo usuário de acordo com sua preferência, desde que seja negativo. O uso de -1 é funcionalmente equivalente ao uso de -2, -3, etc.

A única consideração que deve ser tomada é a de se certificar de sempre armazenar o tipo pré-determinado de informação no respectivo sub-tipo de BLOB. O Firebird não faz nenhuma análise dos dados que estão sendo gravados, portanto essa é uma responsabilidade do aplicativo. Nenhum erro será retornado pelo Firebird se um tipo errado de dado for inserido em um BLOB de sub-tipo incorreto, mas um aplicativo pode ser prejudicado se ao recuperar as informações do BLOB, a mesma não corresponder ao formato esperado.

Sintaxe :

Estas declarações é na criação da tabela :

```
MEMO BLOB SUB_TYPE 1;  
FOTO BLOB SUB_TYPE 0;
```

```
CREATE TABLE FUNCIONARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL,  
.....  
FOTO BLOB SUB_TYPE 0,  
EXPERIENCIA BLOB SUB_TYPE 1 SEGMENT SIZE 80,  
)
```

Na criação da tabela acima, você verificou uma nova informação "SEGMENT".

Abaixo a explicação :



O tamanho de segmento (Segment Size), é um pequeno pedaço de informação, tipo um conselho, que é mantido com a definição de um Blob. Quando se abre um Blob, pode-se perguntar por segmentos de qualquer tamanho, mas para alguns Blobs um determinado tamanho é mais conveniente que um outro. Blobs que armazenam texto, por exemplo, freqüentemente utilizam segmentos de tamanho 80. O pré-processador e outros programas utilitários usam o tamanho do segmento para determinar o tamanho de buffers que são necessários para transferência de dados para e do Blob.

3.2. **CHAR(N)**

O tipo de Dado CHAR, tem o seu tamanho definido na hora da criação da tabela. Seu tamanho máximo é de 32767, 32k. Este tipo tem o seu tamanho fixo.

```
CREATE TABLE FUNCIONARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL,  
...  
SEXO CHAR(01)  
)
```

Este tipo de dado é usado quando você realmente souber o tamanho da coluna/campo a ser criada. Outro exemplo, é criar o coluna de CNPJ, DOMAIN BOLLEAN.

3.3. **VARCHAR(N)**

O tipo de Dado VARCHAR, tem o seu tamanho definido na hora da criação da tabela. Seu tamanho máximo é de 32767, 32k. Este tipo tem o seu tamanho variado na tabela. Isto é, se você criar uma coluna de 45 Caracteres, mas, a coluna tenha apenas 20 Caracteres gravados, o restante, os 25 Caracteres são descartados.

```
CREATE TABLE FUNCIONARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL  
)
```

Este tipo de dado é usado quando você realmente não souber o tamanho da coluna/campo a ser criada. Outros exemplos, são criar campos Descrições, Inscrições Estaduais.

3.4. **DATE**

O tipo de Dado DATE, no DIALECT 3, armazena a Data, e seu tamanho é de 32 bits inteiros longos. O Tipo Date, no Dialect 1, armazena Data e Hora ao mesmo tempo.

```
CREATE TABLE FUNCIONARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL,  
...  
DATA_ADMISSAO DATE  
)
```

3.5. **TIME**

O tipo de Dado TIME, no DIALECT 3, armazena a hora, e seu tamanho é de 32 bits inteiros longos. O Tipo Time, no Dialect 1, não existia.

```
CREATE TABLE FUNCIONARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
NOME VARCHAR(50) NOT NULL,  
...  
DATA_ADMISSAO DATE  
)
```



```
....  
HORA_ENTRADA TIME,  
HORA_SAIDA TIME  
)
```

3.6. **TIMEStamp**

O tipo de Dado **TIMEStamp**, no DIALECT 3, armazena a Data e a hora ao mesmo tempo, e seu tamanho é de 32 bits inteiros longos. O Tipo **TimeStamp**, no Dialect 1, não existia.

```
CREATE TABLE PRODUTOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
....  
DATA_HORA_MOVIMENTACAO TIMESTAMP  
)
```

3.7. **DECIMAL**

O tipo de Dado **DECIMAL**, armazena dígitos a serem gravados na precisão especificada na criação da tabela.

```
CREATE TABLE FUNCIOARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
....  
SALARIO DECIMAL(15,02)  
)
```

3.8. **NUMERIC**

O tipo de Dado **NUMERIC**, armazenas dígitos a serem gravados na precisão especificada na criação da tabela.

```
CREATE TABLE FUNCIOARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
....  
SALARIO NUMERIC(15,02)  
)
```

3.9. **SMALLINT**

O tipo de Dado **SMALLINT**, armazena dígitos a serem gravados, mas, com o limite de : -32768 a 32767. Serve para armazenar dados numéricos pequenos.

```
CREATE TABLE FUNCIOARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
....  
ALTURA SMALLINT  
)
```

3.10. **INTEGER**

O tipo de Dado **INTEGER**, armazena dígitos a serem gravados, mas, diferente do **SMALLINT**, não existe um limite aparentemente, este tipo é de 32 bits, tem a escala de valores em : -2.147.483.648 até 2.147.483.648

Ex :

```
CREATE TABLE FUNCIOARIOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
...)
```



3.11. FLOAT

O tipo de Dado FLOAT, armazena dígitos a serem gravados, mas, com precisão simples de 7 dígitos.

```
CREATE TABLE PRODUTOS (  
ID INTEGER NOT NULL PRIMARY KEY,  
DESCRICAO VARCHAR(50) NOT NULL,  
VLR_ULT_CMP_ITEM FLOAT  
...)
```

3.12. DOUBLE PRECISION

Este é o tipo de campo no qual eu recomendo para uso monetário/valores no Firebird, Dialect 3. Sua precisão é de 64 bits, na documentação fala em usar apenas para valores científicos, mas, eu o uso em todos os sistemas, e obtenho sempre o arredondamento e precisão desejada.

```
CREATE TABLE MOV_FINANCEIRA (  
IDCODMOEDA CHAR(03) NOT NULL PRIMARY KEY,  
DATA_MOVIMENTACAO DATE NOT NULL PRIMARY KEY,  
VALOR_TOTAL_MOVIMENTACAO_DIA DOUBLE PRECISION  
);
```



4. STORED PROCEDURES E TRIGGERS NO FIREBIRD

4.1. INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 01)

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 05], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

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 [FIRE 09].

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, otimizadas 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 otimizada 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 fazer-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 NomeProcedure
<parâmetros de entrada>
RETURNS
<parâmetros de saída>
AS
<declaração de variáveis locais>
BEGIN
<comandos da procedure>
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 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.

4.2. INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 02)

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 06], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

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 caractere 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  
    /*
```

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



* 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.

4.3. INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 03)

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 07], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

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	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 (:mgr_no IS NULL) THEN
BEGIN
mgr_name='--TBH--';
title="";
END
ELSE
SELECT full_name, job_code
FROM employee
WHERE emp_no=:mgr_no
INTO :mgr_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 várias 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 BEGINEND.

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;
```

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



```
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  
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 está 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.

4.4. INTRODUÇÃO A STORED PROCEDURES NO FIREBIRD (PARTE 04)

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 08], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

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 estes 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 linha abaixo mostra esta procedure sendo chamada usando um comando SELECT. Observe que uma linha e uma coluna serão retornadas e o nome da coluna é o nome do parâmetro de saída.

```
SELECT * FROM CUSTOMER_COUNT
```

Você também pode chamar procedures que necessitem de parâmetros de entrada a partir do IB Console. Ex:



```
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 ^
```

EXECUTANDO UMA SELECT PROC. COM PARÂMETROS DE ENTRADA

O parâmetro de entrada, CUST_NO, é passado para a procedure 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
```

O código a seguir foi retirado do evento “onclik” de um botão “Execute” de um formulário em Delphi 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;
```

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.

4.5. TRIGGERS

Este capítulo foi extraído do artigo de Rodrigo Cardoso [FIRE 10], para fins de pesquisa e desenvolvimento nas aulas de Banco de Dados.

Triggers ou **Gatilhos** 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 (SEQUENCIAIS)

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 apresenta-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;
```

RESUMO IMPORTANTE

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.

5. ANEXOS (DICAS)

5.1.1. EXCLUIR CÓDIGO-FONTE DE STORED PROCEDURE

Uma grande preocupação que tem tomado conta da cabeça de muitos programadores é a possibilidade de um programador concorrente pegar o código-fonte das stored procedures armazenadas em banco de dados InterBase/FireBird. Uma solução encontrada é apagar o código-fonte diretamente da tabela de sistema onde o InterBase grava as informações relativas às stored procedures. Para fazer isto execute o comando abaixo:

```
UPDATE RDB$PROCEDURES SET RDB$PROCEDURE_SOURCE = 'empty'
```

Observações

A mesma coisa pode ser feita com triggers. No entanto é importante lembrar que você não deve atribuir NULL, pois havia um bug no InterBase que fazia o trigger ser disparado duas vezes caso o código-fonte estivesse NULL. Não sei se o bug foi corrigido. De qualquer forma, atribua uma string qualquer, tal como no exemplo acima.

5.1.2. OBTER OS CAMPOS DA CHAVE-PRIMÁRIA

Execute o comando SELECT abaixo para obter os nomes dos campos da chave-primária de uma tabela do InterBase ou FireBird.

```
SELECT RDB$FIELD_NAME
FROM
  RDB$RELATION_CONSTRAINTS C,
  RDB$INDEX_SEGMENTS S
WHERE
  C.RDB$RELATION_NAME = 'NomeDaTabela' AND
  C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY' AND
  S.RDB$INDEX_NAME = C.RDB$INDEX_NAME
ORDER BY RDB$FIELD_POSITION
```

Observações

Estes objetos com nomes iniciados com RDB\$ são chamados de objetos de sistema e são usados internamente pelo InterBase/FireBird. As tabelas começadas com RDB\$ contém dados sobre a estrutura do banco de dados.

5.1.3. OBTER A DATA DO SERVIDOR

Isto é facilmente possível se você usa um banco de dados Client/Server, tal como Interbase, SQL Server, Oracle, etc. No Interbase6 execute a Query abaixo:

```
SELECT CURRENT_DATE FROM RDB$DATABASE;
```

O resultado é a data do servidor onde está rodando o Interbase Server.

Observações

A tabela usada no SELECT foi RDB\$DATABASE, mas poderia ser qualquer tabela que possua apenas um registro. RDB\$DATABASE é uma tabela de sistema do Interbase.

5.1.4. LISTAR AS TABELAS E VIEWS DO BANCO DE DADOS

Tabelas e views:

```
SELECT RDB$RELATION_NAME FROM RDB$RELATIONS;
```

Somente tabelas:

```
SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
WHERE RDB$VIEW_BLR IS NULL;
```

Somente views:

```
SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
WHERE NOT RDB$VIEW_BLR IS NULL;
```

Observação:

Para não incluir as tabelas e views de sistema, acrescente o filtro (RDB\$SYSTEM_FLAG = 0 OR RDB\$SYSTEM_FLAG IS NULL) na cláusula WHERE.

5.1.5. BACKUP E RESTORE COM GBAK

O InterBase/FireBird possui uma ferramenta de linha de comando específica para fazer e restaurar cópias de segurança (backup). No Windows o nome do programa é **gbak.exe** e no Linux seu nome é **gbak** (sem extensão). Em ambos os sistemas a localização deste arquivo é o sub-diretório **bin** do InterBase/FireBird. As sintaxes básicas deste comando são:

Elton Ricelli elton.npd@unirondon.br / Rafael Nantes rafael.npd@unirondon.br



=> Para fazer um backup:

```
gbak -b -user usuario -password senha arquivo_banco arquivo_backup
```

=> Para restaurar um backup:

```
gbak -r -user usuario -password senha arquivo_backup arquivo_banco Onde:
```

- **usuario:** é o nome de login do usuário (geralmente SYSDBA).
- **senha:** é a senha do usuário.
- **arquivo_banco:** é o arquivo de banco de dados (geralmente com extensão .gdb).
- **arquivo_backup:** é o arquivo de backup (geralmente com extensão .gbk).

Exemplo de backup e restore respectivamente:

```
gbak -b -user SYSDBA -password masterkey c:\sistema\dados.gdb c:\backup\dados.gbk  
gbak -r -user SYSDBA -password masterkey c:\backup\dados.gbk c:\sistema\dados.gdb
```

5.1.6. CRIAR E USAR DOMÍNIOS (DOMAIN'S)

No InterBase e FireBird domínios são como tipos de dados. Tais domínios têm grande semelhança com o conceito de domínio aplicado à matemática, ou seja, um domínio define um conjunto de valores válidos para uma dada situação. Podemos criar qualquer banco de dados sem fazer uso explícito de domínios. No entanto usar domínios explicitamente pode deixar o banco de dados mais organizado, com regras claras e bem definidas, e ainda conseguir uma economia substancial de mão de obra na construção e manutenção do banco. Para demonstrar a utilidade dos domínios, vamos criar dois exemplos.

Exemplo com uso explícito de domínios:

```
CREATE DOMAIN DM_ChavePrimaria INTEGER NOT NULL CHECK(VALUE > 0);  
CREATE DOMAIN DM_NomePessoa VARCHAR(40) NOT NULL;  
CREATE DOMAIN DM_Fone VARCHAR(20);  
CREATE DOMAIN DM_Renda NUMERIC(9,2) DEFAULT 0 NOT NULL CHECK(VALUE >= 0);
```

```
CREATE TABLE Cliente(  
  Codigo DM_ChavePrimaria,  
  Nome DM_NomePessoa,  
  Fone DM_Fone,  
  Fax DM_Fone,  
  Celular DM_Fone,  
  Renda DM_Renda,  
  CONSTRAINT PK_Cliente PRIMARY KEY(Codigo));
```

Comentários:

- O benefício imediato do uso explícito de domínios é a organização do código que define as tabelas.
- Como um mesmo domínio será usado em várias tabelas (exemplo: **DM_NomePessoa**), ganharemos muito tempo ao definir outras tabelas que comporão o banco de dados.
- O domínio **DM_Fone** é um exemplo que demonstra como um mesmo domínio pode ser usado para colunas diferentes que possuem conteúdos semelhantes.
- Os domínios **DM_ChavePrimaria** e **DM_Renda** mostram aspectos mais interessantes na declaração de domínios, tais como a especificação de um valor padrão (DEFAULT) e regras para validação (CHECK).
- Se mais tarde resolvermos alterar os nomes de pessoas para 50 caracteres, ou seja, VARCHAR(50), bastará alterar a definição do domínio **DM_NomePessoa** e todos os campos definidos com este domínio serão automaticamente ajustados. Neste caso bastaria o comando **ALTER DOMAIN DM_NomePessoa TYPE VARCHAR(50)**.

Nos bancos de dados que crio, uso domínios explicitamente para todos os campos de todas as tabelas, mesmo onde aparentemente são desnecessários. Mas é bom lembrar que domínios mal definidos podem trazer mais prejuízos do que benefícios. Portanto, antes de sair criando domínios deliberadamente, faça um estudo minucioso do banco de dados a ser construído.



6. REFERÊNCIAS BIBLIOGRÁFICAS

[FIRE 01] CARDOSO, Rodrigo. **Você conhece o Firebird?** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=13>> Acesso em 12 out. 2003

[FIRE 02] CARDOSO, Rodrigo. **Ferramentas Administrativas.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=32>> Acesso em 13 out. 2003

[FIRE 03] CARDOSO, Rodrigo. **SQL – Structured Query Language.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=41>> Acesso em 13 out. 2003

[FIRE 04] CARDOSO, Rodrigo. **Tipos de Dados – Firebird.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=51>> Acesso em 13 out. 2003

[FIRE 05] CARDOSO, Rodrigo. **Stored Procedure – Parte 01.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=96>> Acesso em 13 out. 2003

[FIRE 06] CARDOSO, Rodrigo. **Stored Procedure – Parte 02.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=104>> Acesso em 13 out. 2003

[FIRE 07] CARDOSO, Rodrigo. **Stored Procedure – Parte 03.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=113>> Acesso em 13 out. 2003

[FIRE 08] CARDOSO, Rodrigo. **Stored Procedure – Parte 04.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=118>> Acesso em 13 out. 2003

[FIRE 09] TODD, Bill Todd. **Borland Developers Conference San Diego 2000.** São Paulo. 2002. Traduzido e adaptado com autorização do autor por: Alessandro Cunha Fernandes, Comunidade Firebird.

[FIRE 10] CARDOSO, Rodrigo. **Triggers – Firebird.** dez. 2003. Disponível em: <<http://www.infosquad.net/colunas/firebird/index.php?ID=XX>> Acesso em 13 out. 2003