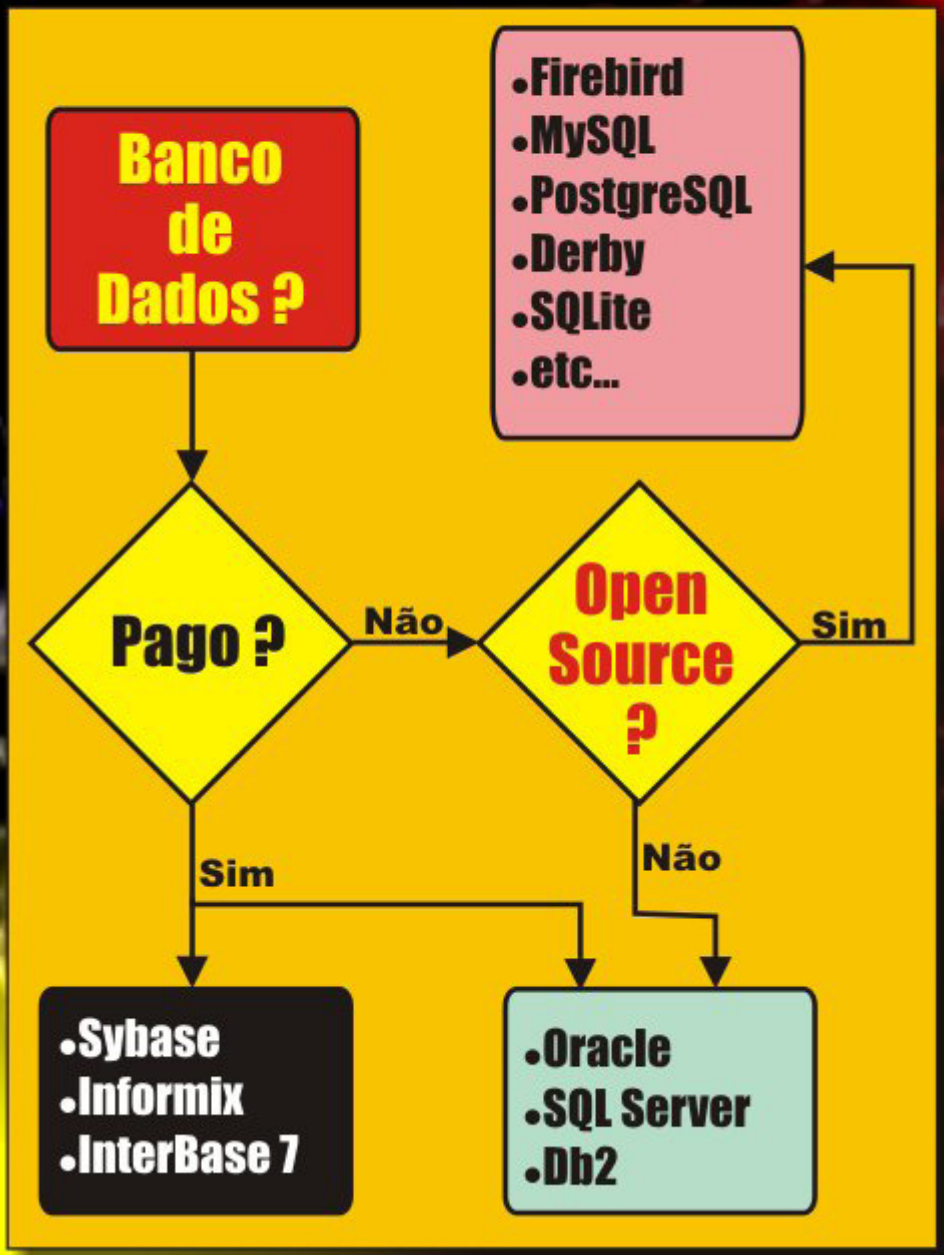
DBFree MAGAZINE

Edição #8

Fevereiro 2006



- ✓ A escolha do Banco de Dados
- ✓ Famosos e gratuitos
- ✓ Normalização e Integridade Referencial
- ✓ Primeiros passos com a ZeosLib
- ✓ Log de alterações no PostgreSQL

Editorial

Bem vindos a primeira edição da DB FreeMagazine do ano de 2006 que, diga-se de passagem, está muito boa, sendo uma das **maiores** edições que já publicamos!

Estamos vivendo uma época bastante especial, no que diz respeito aos bancos de dados. Grandes empresas como Oracle, IBM e Microsoft estão revendo sua política de comercialização dos SGBDs. Muitas já lançaram versões **gratuitas** dos seus bancos; coisa que até pouco tempo atrás seria inimaginável! Nesta edição temos dois artigos que tratam dessa questão, e irão lhe ajudar a escolher um banco de dados, e a entender as licenças de alguns dos novos bancos "free", afinal, não devemos esquecer do famoso ditado: "*quando a esmola é demais, o santo desconfia*".

Temos também um ótimo artigo que trata da aplicação das regras de Integridade através dos recursos nativos na maioria dos SGBDs. Além do conceito teórico envolvido no assunto, que aborda também a normalização dos dados, são mostrados diversos exemplos práticos de código para dois famosos SGBDs: *Oracle* e *Firebird*.

Recebemos alguns emails de pessoas reclamando que a revista não publica material do banco X ou Y. Como vocês devem saber, dependemos da comunidade para nos enviar material para ser publicado. Se a comunidade do banco X ou Y não nos ajuda, infelizmente continuarão ficando de fora das nossas edições. A DB FreeMagazine é uma revista gratuita que trata de todo e qualquer SGBD relacional, seja ele comercial, gratuito e/ou Open Source. O espaço está aberto para quem quiser contribuir!

Boa leitura a todos!

Carlos H. Cantu
Fevereiro/2006

DB Free MAGAZINE

Informações

DB FreeMagazine nº 008 - Ano II
Fevereiro/2006
Contato geral:
webmaster@dbfreemagazine.com.br

Equipe editorial

Carlos H. Cantu
(cantu@dbfreemagazine.com.br)
Luiz Paulo de Oliveira Santos
(lpaulo@dbfreemagazine.com.br)

Contribuíram nessa edição

- Carlos H. Cantu
- Luiz Paulo de Oliveira Santos
- Mauro Henrique Costa Matos
- Robert Nunes
- Mário Lúcio Gomes de Faria

Dica para melhor visualização

Utilize a resolução **1024x768** pixels e configure o *Acrobat Reader* para **Zoom** de **100%**. Feche todas as abas laterais e esconda as barras de ferramentas, liberando o máximo de área útil na tela, ou simplesmente rode a revista em modo *fullscreen*. Usuários de **Linux**: Recomendamos utilizar o **Acrobat Reader** para Linux a fim de garantir 100% de compatibilidade com o formato.

Saindo do Forno...

Firebird 2.0 beta2

Acaba de sair a versão beta 2 do Firebird 2.0. É esperado que a versão final do software seja lançada até a metade do ano. A versão beta 2 corrigir várias falhas que foram detectadas após o lançamento do beta 1.

Fonte: *www.firebirdsql.org*

Lançado PostgreSQL 8.1.3

A versão 8.1.3 corrige uma séria vulnerabilidade que existia em todas as versões anteriores do PostgreSQL 8.1. A falha permitia que um usuário do banco de dados aumentasse os poderes do ROLE utilizado.

Fonte: *http://www.postgresql.org/about/news.476*

Borland vendendo sua linha de IDEs

Um anúncio recente surpreendeu muitos desenvolvedores: A Borland, fabricante do Delphi e do JBuilder, pretende vender sua linha de IDEs (que inclui ambos os produtos e vários outros produtos relacionados, como o próprio InterBase) e irá se dedicar apenas as ferramentas de ALM (Application Lifecycle Management). Ainda não se tem um comprador, mas a equipe de desenvolvimento diz que ele será escolhido “a dedo”.

Fonte: *http://www.borland.com/us/company/news/Tod_Nielsen_customer_shareholder_letter_02-08-06.html*

Compatibilidade do Firebird com o SQL:2003

Foi publicado recentemente um artigo que mostra a compatibilidade do Firebird 1.5 e 2.0 com o padrão SQL:2003. O artigo foi escrito por Dmitry Yemanov, integrande da equipe de desenvolvedores do Firebird, e estará sendo constantemente atualizado.

Fonte: *http://www.firebirdsql.org/index.php?op=devel&sub=engine&id=SQL_conformance&nosb=1*

DB2 Express-C

A IBM anunciou uma versão gratuita do seu banco de dados DB2, na intenção de atrair novos desenvolvedores.

Fonte: *http://news.zdnet.com/2100-3513_22-6032676.html*

Novo site de notícias sobre Firebird

Em pouco mais de um mês após ter sido lançado, o site FirebirdNews.org já se tornou a melhor fonte de informação sobre notícias e novidades referente ao banco de dados Firebird.

Fonte: *www.firebirdnews.org*

Oracle disponibiliza correções

A Oracle disponibilizou recentemente correções para 82 vulnerabilidades que afetavam seu banco de dados e servidores de aplicação.

Fonte: *http://www.computerworld.com/databasetopics/data/software/story/0,10801,107825,00.html*

Firebird no anuário InfoCorporate

A edição 2006 do Anuário InfoCorporate traz uma matéria que cita o Firebird. A FireBase, que oferece consultoria e suporte especializado para o Firebird no Brasil, também foi citada.

Fonte: http://www.firebirdnews.org/?p=103

Investimento de mais de US$ 1mi

Algumas empresas receberão um investimento de mais de um milhão de dólares do departamento americano de segurança Homeland, para rastrear falhas de segurança nos códigos de produtos Open Source. O MySQL e o Firebird estão entre eles.

Fonte: http://www.computerweekly.com/Articles/2006/01/11/213616/USfinancesopen-sourcebughunt.htm

Oracle compra Sleepycat

A Oracle anunciou a compra da Sleepycat, que desenvolve o BerkeleyDB utilizado no MySQL. Recentemente a Oracle havia comprado a Innobase, fabricante do InnoDB, também utilizado no MySQL.

Fonte: *http://news.zdnet.com/2100-3513_22-6039070.html*

Jim Starkey vai para MySQL AB

A MySQL AB acaba de comprar a Netfrastructure, empresa de Jim Starkey (criador do InterBase).

Com a venda, Jim também trabalhará *fulltime* para a MySQL. Ann Harrison (IBPhoenix), esposa de Jim, fará algum trabalho para a MySQL, mas continuará envolvida com o Firebird.

Fonte: *http://www.firebirdnews.org/?p=128*

Open Source - o único e verdadeiro modo de desenvolver software

"*As verdadeiras revoluções, aquelas que persistem, são geralmente as mais quietas*". Esse é o início de um artigo interessante publicado no site Linux-Watch, que trata de algumas considerações sobre a atual "revolução" Open Source.

Fonte: *http://www.linux-watch.com/news/NS3670538334.html*

Conceitos de Normalização e Integridade Referencial

por Robert Nunes

A utilização de softwares associados a bancos de dados nas empresas é uma realidade. Contudo, existe a preocupação quanto à confiabilidade dos dados armazenados. Para isso, existem meios de garantir que os dados sejam armazenados corretamente, a fim de preservar a integridade do Banco de Dados.

As regras de restrições de integridade visam manter os dados de forma organizada e qualitativa, evitando inconsistências, atuando como um resguardo contra operações acidentais ou propositais, que possam causar inconsistências nos mesmos.

Inicialmente, é importante diferenciar os termos "segurança" e "integridade", pois no contexto de banco de dados, tais conceitos são comumente confundidos, embora sejam completamente distintos.

O termo "integridade" refere-se à precisão ou validação dos dados, garantindo que as operações sejam feitas de forma correta e que os dados sejam normalizados; enquanto que "segurança" trata da garantia contra invasão e até mesmo contra acessos não autorizados de usuários.

Existem várias formas de se garantir a Restrição de Integridade, entre elas:

- Integridade referencial, restrição de chaves (primárias, candidatas e estrangeiras), restrições n*ot null, checks, constraints*;
- Gatilhos (t*riggers), stored procedures;*

OBJETIVOS

No desenvolvimento deste artigo, foram analisados dois SGBDs para a aplicação de exemplos: **Oracle** e **Firebird**. O primeiro é de grande porte e proprietário, muito utilizado por grandes corporações pela versatilidade e segurança que oferece para os DBAs. Com o crescimento do mercado de grandes softwares, o banco de dados ORACLE vem se consolidando como uma das melhores ferramentas para desenvolvimento de banco de dados.

O Firebird conta com uma grande base de usuários, oferecendo segurança e recursos, além de ser uma ferramenta de código aberto e gratuita. É amplamente utilizado no Brasil, principalmente pelos usuários das ferramentas de programação da Borland, como por exemplo o Delphi.

Os exemplos apresentados no decorrer desse artigo estarão codificados para ambos os bancos de dados.

1. CONCEITOS E NOÇÕES SOBRE INTEGRIDADE DE DADOS

Através da normalização dos dados e da aplicação de regras de integridade, podemos "proteger" as informações contra mudanças inconsistentes, gerando exceções (erros) quando alguma operação "quebrar" uma regra de integridade.

1.1. Restrição de Integridade

A *Restrição de Integridade* é uma regra que deve ser seguida e obedecida por todos os estados do banco de dados consideradas consistentes e confiáveis. As regras de integridade fornecem a garantia de que as mudanças feitas no Banco de Dados não resultem na perda da consistência dos mesmos. (DATE – 1990).

Existem inúmeras formas de se garantir a integridade dos dados. A principal recomendação é que os dados estejam **normalizados**. Para que isso aconteça, é necessário que sejam aplicadas as *formas de normalização* - conjuntos específicos de limitações que devem ser satisfeitos. (DATE – 1990).

As restrições de integridade resguardam o sistema contra transações acidentais ou propositais, assegurando que as mudanças feitas não levem a causar inconsistência dos dados.

O aspecto "integridade de dados" pode ser considerado como o mais importante dentro de uma estrutura de banco de dados, já que não adianta criar uma estrutura ágil e de fácil utilização, se as informações contidas nela forem frágeis e de baixa confiança.

A seguir, serão apresentadas as regras normais para a aplicação da normalização dos dados.

1.2 Formas Normais

A **normalização** é um dos principais requisitos para que possa ser aplicada a restrição de integridade. Ela faz com que os arquivos de dados, tornem-se mais adequados para o armazenamento e atualização, evitando problemas provocados por falhas no projeto. *A normalização consiste em eliminar elementos de dados redundantes e grupos repetitivos* (MARQUES – 1995).

A maioria dos processos de normalização consiste, basicamente, em regras que definem de maneira adequada os elementos de dados que serão utilizados na base de dados. O objetivo da normalização é principalmente evitar as anomalias; a solução para essa situação é a decomposição de uma relação em uma ou mais relações, baseando-se nas regras de normalização. Tal situação pode não ser benéfica, do ponto de vista de performance; mas, por outro lado, a garantia na consistência das informações é um grande benefício.

O processo de normalização consiste na aplicação de cinco regras. Porém, na maioria dos casos, são aplicadas somente as **três** primeiras regras, suficientes para manter uma estrutura organizada de forma que se possa aplicar as restrições de integridade de maneira satisfatória (ver ***figura 1***).

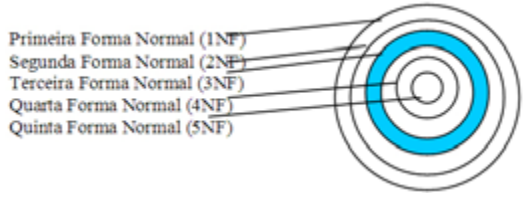


Figura 1: Representação da Normalização.

Vejamos as cinco formas normais:

1ª Forma Normal: consiste em abstrair das estruturas os elementos repetitivos, ou seja, aqueles dados que podem compor uma estrutura de vetor. A aplicação da primeira regra consiste em remover esses elementos repetitivos;

2ª Forma Normal: trata-se de retirar da estrutura que possui chaves compostas, os elementos que são funcionalmente dependentes de parte da chave. Pode-se garantir que a estrutura está na 2ª forma normal, se estiver na 1ª forma normal e não possuir campos que são funcionalmente dependentes de parte da chave;

3ª Forma Normal: consiste em retirar os campos que são funcionalmente dependentes dos outros campos que não sejam chave; a estrutura está na 3ª forma normal, se estiver na 2ª forma normal e não possuir campos dependentes de outros campos não-chaves;

4ª Forma Normal: essa regra se aplica as estruturas que mantêm relacionamentos ternários ou superiores, e se detecta dependências funcionais multivaloradas, ou seja, um ou mais atributos determinam vários valores de um outro atributo. Para efeito de analise, são levadas em conta somente as estruturas com chave formada por uma tripla ou mais, que não tenham atributos não-chave. Normalmente, é detectada a aplicação da 4º forma normal quando se tem mais de um atributo multivalorado em uma tabela não normalizada.

5ª Forma Normal: essa regra também é aplicada em estruturas que mantém relacionamentos ternários sem atributos não-chave, a tabela está na 5ª forma normal se um relacionamento triplo puder ser decomposto em 3 tabelas, sem que gere dados incorretos quando os mesmos forem combinados novamente em uma tabela de relacionamento triplo.

1.3 Desempenho em Banco de Dados

A palavra desempenho pode ser definida como agilidade ou, na maioria das vezes, como velocidade em que os dados são manipulados e recuperados na base, ou seja, uma das formas de saber a quantidade de informações que o SGBD consegue processar em um determinado tempo.

A indexação é uma das técnicas, ou método de acesso, que visa a melhoria do desempenho na recuperação de informações. O sucesso para se obter um resultado satisfatório em uma determinada base de dados, deve ter como principal requisito o nível físico. Assim, o sistema saberá quando operar sobre essas estruturas.

1.4 Desempenho / Integridade

Em uma base normalizada, normalmente ocorrem buscas de informações por vários acessos e verificações, deixando assim as consultas mais lentas. Conforme a base de dados cresce, a complexidade dos acessos segue a mesma proporção, o que pode acarretar maior demora na obtenção das informações desejadas, tornando um projeto inviável para uma determinada aplicação.

Um dos aspectos que deve ter um tratamento especial é a redundância dos dados. Apesar de fugir das regras das formas normais, uma pequena redundância pode agilizar muito uma determinada pesquisa.

1.5 Integridade *vs.* Segurança

Como visto anteriormente, segurança refere-se à proteção de dados contra a divulgação, alteração ou destruição não autorizada; enquanto "integridade" refere-se à precisão ou validade dos dados. Ou seja, segurança garante aos usuários permissão para executar certas operações; integridade assegura que as operações sejam feitas de forma correta.

1.5.1 Algumas funções oferecidas pela restrição de integridade:

a) Bloqueio contra a redundância de dados deixando a base de dados mais ágil e de fácil manipulação.
b) Base de dados consistente e flexível, trazendo diversos benefícios para uma produção ágil e segura, facilitando a vida dos programadores e gerentes de software, proporcionando uma maior performance e qualidade no desenvolvimento.
c) O resultado final da base de dados, com a aplicação das regras de integridade, agiliza possíveis processos de reestruturação.

1.5.2 Medidas ou políticas de segurança

a) *Física*: proteção física do local onde estão localizados os softwares ou sistemas computacionais, para evitar a entrada indevida de usuários não autorizados e, assim, garantir a integridade dos equipamentos.
b) *Humana*: deve cuidar das autorizações de acesso ao usuário de sistema computacional.
c) *Sistema Gerenciador de Banco de Dados (SGBD)*: deve cuidar das autorizações dos acessos a dados dos sistemas computacionais.
d) *Sistema Operacional (SO)*: deve manter sempre o SO seguro contra acessos não autorizados.

1.6 Integridade Referencial

Integridade Referencial é um dos principais mecanismos para a garantia da integridade dos dados. A principal função é garantir o relacionamento entre tabelas, fazendo com que informações interdependentes sejam consistidas, como ilustrado na **figura 2**.

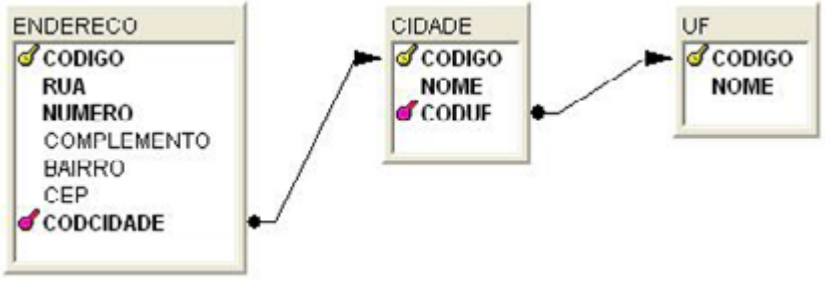


Figura 2: DER Integridade referencial.

Pode ser aplicada a integridade referencial na criação de estrutura, declaração de chaves e criação de *constraints,* entre outros. No exemplo da **figura 2**, através da integridade referencial, podemos garantir que se o campo código da tabela UF for alterado, essa mudança será propagada para as tabelas relacionadas em cascata, atualizando todas as referências. Essa ação é conhecida como "*propagar atualizações dos campos atualizados*". Analogamente, existe a exclusão em cascata, denominada "*propagar exclusão dos registros relacionados*".

2. Mecanismos para garantia de Restrição de Integridade

Com a evolução dos sistemas gerenciadores de banco de dados, foram desenvolvidas varias técnicas e mecanismos para a aplicação das restrições de integridade, provocando o aumento no número de ferramentas de gerenciamento de banco de dados que facilitam a implementação das mesmas em projetos de grande, médio e pequeno porte.

2.1 Regras de Normalização

As regras de normalização são de grande importância no quesito restrição de integridade. O primeiro passo para se conseguir integridade em uma base de dados é a aplicação das três principais formas normais. Aplicando-se tais regras à base de dados, teremos os seguintes efeitos:

- Evitaremos anomalias, equívocos de atualização e inconsistência dos dados;
- Uma possível leve degradação de desempenho, devido ao aumento do número de leituras necessárias para a recuperação das informações desejadas;
- Proporcionar um menor impacto nas gravações, o qual tende a ser, de qualquer maneira, um retardo de processo;
- Garantir uma base de dados consistente, facilitando o projeto e evitando problemas futuros contra falhas de projetistas;

***Nota**: É comum a não aplicação de normalização em base de dados somente leitura, nos quais o alto desempenho é necessário (ex: Data Warehouse e base de dados de consulta para Web).*

Para uma normalização eficiente, é primordial que seja definido um campo chave para a estrutura. Tal campo permitirá identificar os demais campos da estrutura. A seguir, exemplificaremos parte de um projeto, em sua forma desnormalizada. Em seguida, aplicaremos as regras de normalização.

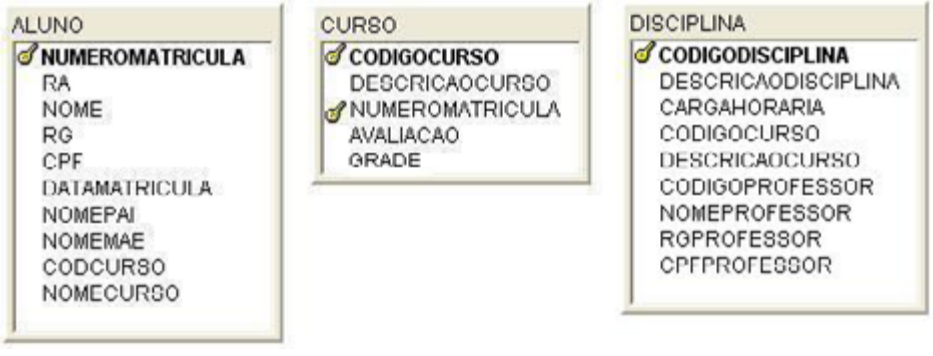


Figura 3: Estrutura não normalizada.

Observando a **figura 3**, nota-se que as tabelas ALUNO, CURSO e DISCIPLINA não estão normalizadas. Com a aplicação das principais regras, elas serão normalizadas, aumentando a quantidade de tabelas.

2.1.1 1ª Forma normal

A tabela se encontra na primeira forma normal quando seus atributos não contêm grupos de repetição. Assim, podemos ver que a tabela ALUNO não está de acordo com a primeira forma normal.

Considere que um casal tenha mais de um filho. Para todo filho que entrar na faculdade, será digitado o nome do pai e nome da mãe novamente, formando assim um grupo de repetição. Além disso, pode acontecer erros de digitação e os nomes serem digitados de maneiras diferentes, o que pode acarretar problemas com pesquisa ou emissão de relatórios.

Esse problema ocorre porque estão misturados assuntos em uma mesma tabela. Se forem colocados os dados do aluno e dos pais em tabelas diferentes, tal problema será solucionado por meio de um relacionamento do tipo *um para vários*, ou seja, um casal de pais pode ter vários filhos. A **figura 4** mostra essa solução.

A tabela ALUNO foi decomposta, resultando em uma nova tabela (PAI), evitando-se a repetição de informações.

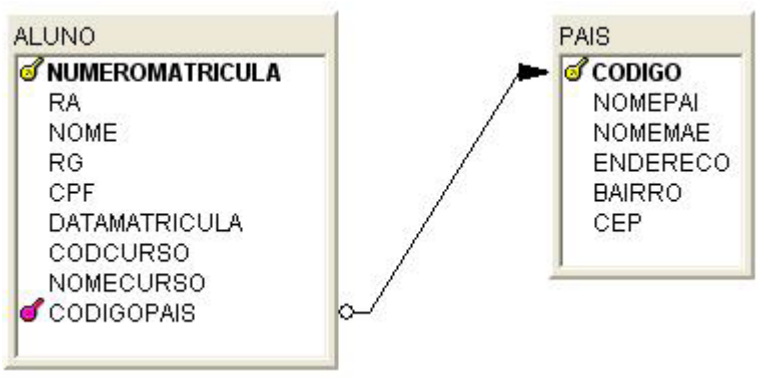


Figura 4: 1ª forma normalizada

Figura 7: 3ª forma não normalizada.

2.1.2 2ª Forma Normal

Uma entidade está na segunda forma normal se estiver na primeira, e todos os seus atributos dependem totalmente da chave primária composta. Se algum campo depender somente de uma parte da chave composta, então esse campo deve pertencer a uma outra tabela.

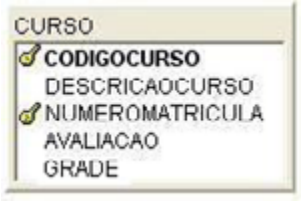


Figura 5: 2ª forma não normalizada.

A chave primária composta é formada pela combinação dos atributos NUMEROMATRICULA e CODIGOCURSO. O campo avaliação depende tanto do atributo NUMEROMATRICULA quanto do CODIGOCURSO. Porém, o atributo DESCRICAOCURSO depende somente do CODIGOCURSO, ou seja, com o código do curso, é possível localizar os dados do curso independentemente do atributo NUMEROMATRICULA. Com isso, há um atributo que faz parte da chave primária e depende apenas de um dos campos que compõem a chave composta, dessa forma pode-se inferir que a tabela da **figura 5** não está normalizada.

A solução para esse problema é bastante simples: aplicando-se a segunda forma normal são criadas duas tabelas, conforme **figura 6**.

2.1.3 3ª Forma normal

Para estar na terceira forma normal, a entidade já deve estar na segunda forma, e não pode possuir nenhum atributo que dependa de outro atributo que não faça parte da chave primária. Podemos ver na **figura 7** que a tabela ALUNO contém um campo NOMECURSO que depende do atributo CODCURSO, que por sua vez não faz parte da chave primária, portanto esta tabela não se enquadra na terceira forma normal. Do mesmo modo, na tabela DISCIPLINA, os campos NomeProfessor, RGProfessor, RGProfessor e CPFProfessor dependem do campo CODIGOPRODFESSOR.

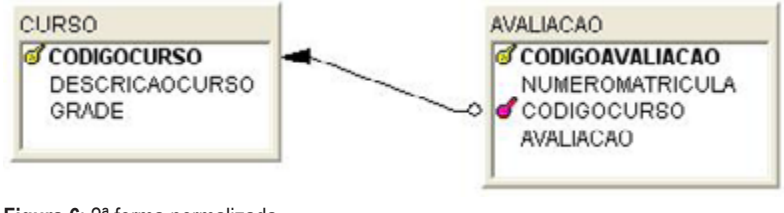


Figura 6: 2ª forma normalizada.

DescricaoCurso depende de CodigoCurso. Nem CodigoCurso, nem CodigoProfessor faz parte da chave primária, portanto a tabela DISCIPLINA também não está normalizada.

A **figura 8** apresenta a versão normalizada do problema exposto acima. Foram criadas duas novas tabelas: CURSO e PROFESSOR, cada uma com seus campos dependentes, e criadas relações de referência entre as tabelas.

Com essas três principais formas normais, pode-se concluir que, por meio do resultado do processo de normalização, a sua aplicação obterá um número maior de tabelas, porém sem problema de redundância dos dados.

É comum, após a aplicação das regras de normalização, que algumas tabelas acabem divididas em duas ou mais. No final do projeto, o número de tabelas é bem maior do que a estrutura original.

Esse processo causa a simplificação dos atributos de uma determinada tabela, colaborando, significativamente, para a estabilidade do modelo de dados usado e reduzindo, consideravelmente, as necessidades de manutenção do mesmo. A **figura 9** mostra a versão normalizada da estrutura inicial (apresentada na **figura 3**).

A versão final do sistema poderá sofrer alguma alteração, para atender às

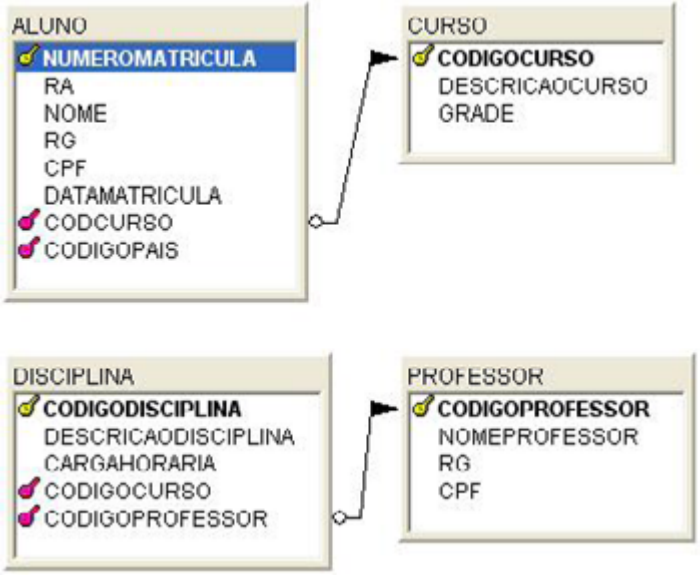


Figura 8: 3ª forma normalizada.

necessidades específicas de cada aplicação, a critério do analista de desenvolvimento durante o projeto físico do mesmo.

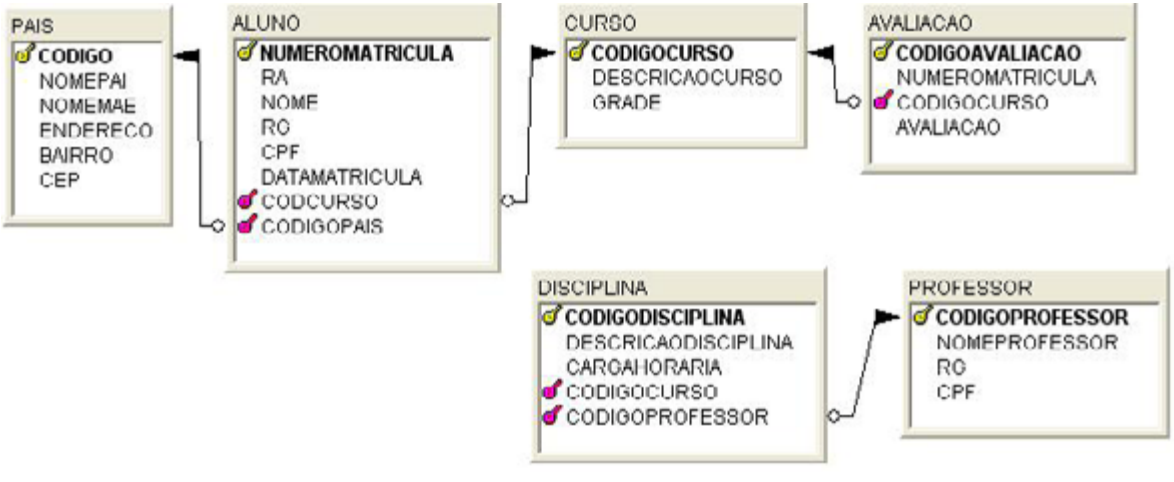


Figura 9: Estrutura normalizada.

2.2 Restrição de *Primary key* (Chave Primária)

A restrição de chave primária, tecnicamente, é a união da restrição de unicidade com a restrição não-nula. Essa restrição indica que a coluna, ou grupo de colunas, pode ser utilizada como identificador único, não podendo aceitar valores repetidos, servindo como ponto de referência para a tabela. Somente **uma** chave primária pode ser especificada em cada tabela.

A criação de uma chave primária implica na definição de uma *constraint* que a identifica, facilitando uma eventual modificação. Quando a *constraint* é criada na tabela, um índice é automaticamente originado sobre as colunas que fazem parte da chave primária. Esse índice não pode ser apagado diretamente; é preciso apagar a *constraint* de *primary key* para que o índice seja removido.

O exemplo abaixo mostra o comando para criação de uma chave primária no campo CODIGO da tabela UF:

```
1. FIREBIRD
2. CREATE TABLE UF (
3. CODIGO CHAR (2) NOT NULL,
4. NOME VARCHAR (40) NOT NULL,
5. CONSTRAINT UF_CODIGO_PK
                PRIMARY KEY (CODIGO))
```

```
1. ORACLE
2. CREATE TABLE UF (
3. CODIGO CHAR (2) NOT NULL,
4. NOME VARCHAR2 (40) NOT NULL,
5. CONSTRAINT UF_CODIGO_PK
                PRIMARY KEY (CODIGO))
```

Além das chaves primárias convencionais, existem as chaves primárias compostas. Estas são formadas por mais de um atributo da tabela. O exemplo abaixo cria uma chave primária composta por dois campos: LOGIN e SENHA, na tabela USUARIO.

```
1. FIREBIRD
2. CREATE TABLE USUARIO (
3. LOGIN VARCHAR (10) NOT NULL,
4. SENHA VARCHAR (10) NOT NULL,
5. CONSTRAINT USUARIO_LOGINSENHA_PK
                PRIMARY KEY (LOGIN,SENHA))
```

```
1. ORACLE
2. CREATE TABLE USUARIO (
3. LOGIN VARCHAR2 (10) NOT NULL,
4. SENHA VARCHAR2 (10) NOT NULL,
5. CONSTRAINT USUARIO_LOGINSENHA_PK
                PRIMARY KEY(LOGIN,SENHA))
```

2.3 Restrição de *Foreign Key* (Chave Estrangeira)

A restrição de chave estrangeira é usada para relacionar tabelas diferentes. Ela especifica que um grupo de um ou mais atributos de uma tabela

referencia a chave primária de uma outra tabela.

A exemplo da restrição de chave primária, a chave estrangeira é criada através de uma *constraint,* originando um índice automático sobre o atributo que a define. Esse índice não pode ser apagado diretamente, sem que a *constraint* também seja apagada.

O exemplo abaixo demonstra a criação de uma chave estrangeira em uma tabela já existente:

```
1. FIREBIRD/ORACLE

2. ALTER TABLE GRPPRODUTO
3. ADD CONSTRAINT
     GRPPRODUTO_CODPRODUTO_FK
     FOREIGN KEY(CODPRODUTO) REFERENCES
     PRODUTO(CODIGO)
```

No exemplo, foi adicionada a chave estrangeira ao atributo CODPRODUTO, da tabela GRPPRODUTO. Ou seja, o atributo CODPRODUTO, da tabela GRPPRODUTO é relacionado ao atributo CODIGO da tabela de PRODUTO, garantindo que todos os grupos de produtos terão seus produtos referenciados de forma correta. A *constraint* que definiu o relacionamento teve o nome de "GRPPRODUTO_CODPRODUTO_FK".

A definição de uma chave estrangeira também pode determinar a execução de operações em cascatas, por intermédio das cláusulas **ON DELETE CASCADE** ou **ON UPDATE CASCADE**. A primeira faz com que quando um registro referenciado por uma chave estrangeira seja apagado, todos os registros que o referenciam também sejam apagados. No exemplo acima, se um determinado produto fosse apagado na tabela PRODUTO, todos os registros da tabela GRPPRODUTO que referenciam o produto apagado seriam também removidos. A cláusula ON UPDATE CASCADE faz com que a alteração do valor do atributo referenciado seja propagada a todos os registros que o referenciam, ou seja, quando o código de um produto for alterado na tabela PRODUTO, todos os registros da tabela GRPPRODUTO que referenciam este produto serão atualizados com o novo valor.

Abaixo temos alguns exemplos da criação de chaves estrangeiras fazendo uso de ON DELETE e ON UPDATE.

```
1. FIREBIRD/ORACLE
2. ALTER TABLE GRPPRODUTO
3. ADD CONSTRAINT GRPPRODUTO_CODPRODUTO_
   FK
4. FOREIGN KEY(CODPRODUTO) REFERENCES
   PRODUTO(CODIGO) ON DELETE CASCADE
```

```
1. FIREBIRD/ORACLE
2. ALTER TABLE GRPPRODUTO
3. ADD CONSTRAINT GRPPRODUTO_CODPRODUTO_
   FK
4. FOREIGN KEY(CODPRODUTO) REFERENCES
   PRODUTO(CODIGO) ON UPDATE CASCADE
```

```
1. FIREBIRD/ORACLE
2. ALTER TABLE GRPPRODUTO
3. ADD CONSTRAINT GRPPRODUTO_CODPRODUTO_
   FK
4. FOREIGN KEY(CODPRODUTO) REFERENCES
   PRODUTO(CODIGO) ON UPDATE CASCADE ON
   DELETE CASCADE
```

A seguir vemos os comandos para criação das tabelas mostradas na **figura 10**, onde a definição das chaves estrangeiras já é feita durante a criação da tabela.

```
1. FIREBIRD
2. REATE TABLE GRPPRODUTO(
3. CODIGO SMALLINT NOT NULL,
4. NOME VARCHAR(40) NOT NULL,
5. CODPRODUTO INTEGER NOT NULL,
6. CONSTRAINT GRPPRODUTO_CODIGO_PK
   PRIMARY KEY(CODIGO),
7. CONSTRAINT GRPPRODUTO_CODPRODUTO_FK
     FOREIGN KEY(CODPRODUTO) REFERENCES
     PRODUTO(CODIGO)
     ON UPDATE CASCADE
     ON DELETE CASCADE)
```

```
1. ORACLE
2. CREATE TABLE GRPPRODUTO(
3. CODIGO NUMBER(4) NOT NULL,
4. NOME VARCHAR2(40) NOT NULL,
5. CODPRODUTO NUMBER(6) NOT NULL,
6. CONSTRAINT GRPPRODUTO_CODIGO_PK
   PRIMARY KEY(CODIGO),
7. CONSTRAINT GRPPRODUTO_CODPRODUTO_FK
     FOREIGN KEY(CODPRODUTO) REFERENCES
     PRODUTO(CODIGO)
     ON UPDATE CASCADE
     ON DELETE CASCADE)
```

No exemplo acima é criada uma tabela GRPPRODUTO, que possui os atributos CODIGO, NOME e CODPRODUTO, em que o CODPRODUTO é a chave estrangeira, relacionado com o atributo CODIGO, da tabela PRODUTO.

2.4 Restrição *Not Null* (valores não nulos)

A restrição *Not Null* simplesmente verifica se uma determinada coluna não está vazia (nula). Funcionalmente, equivale a criação de uma restrição de verificação *check*, do tipo *"campo is not null"*. Sua sintaxe é bastante simples e pode ser estabelecida na criação da tabela, conforme mostrado nos exemplos a seguir:

Figura 10: DER chave estrangeira.

```
1. FIREBIRD
2. CREATE TABLE VEICULO(
3. CODIGO SMALLINT NOT NULL,
4. NOME VARCHAR(60) NOT NULL,
5. ANO TIMESTAMP NOT NULL,
6. CONCESSIONARIA VARCHAR(50) NOT NULL,
7. OBSERVACAO VARCHAR(200),
8. CONSTRAINT COMISSAO_CODIGO_PK PRIMARY
   KEY(CODIGO))
```

```
1. ORACLE
2. CREATE TABLE VEICULO(
3. CODIGO NUMBER(4) NOT NULL,
4. NOME VARCHAR2(60) NOT NULL,
5. ANO DATE NOT NULL,
6. CONCESSIONARIA VARCHAR2(50) NOT NULL,
7. OBSERVACAO VARCHAR2(200),
8. CONSTRAINT COMISSAO_CODIGO_PK PRIMARY
   KEY(CODIGO))
```

Nota-se que os atributos CODIGO, NOME, ANO, CONCESSIONARIA, da tabela VEICULO, são campos não nulos, ou seja, devem possuir algum valor definido.

2.5 Restrição *check* (restrição de verificação)

A Restrição *check,* ou restrição de verificação, é o tipo mais genérico de restrição. Ele determina que a especificação de uma coluna esteja de acordo com uma expressão arbitrária. As expressões das restrições de verificações devem envolver a coluna a ser restringida, caso contrário o comando não fará muito sentido. A definição de uma restrição de *check* através de uma *constraint* auxilia na manutenção e na identificação em possíveis mensagens de erro, pois desse modo é possível nomear a restrição de forma clara. No exemplo a seguir, é demonstrada a criação de uma restrição *check* em tabelas previamente existentes:

```
1. FIREBIRD/ORACLE

2. ALTER TABLE BANCO
3. ADD CONSTRAINT CHECK_CONTA
       CHECK(CONTA BETWEEN 1 AND 99999)
```

No exemplo acima, criou-se uma *constraint* por meio da restrição *check* com a seguinte condição: O atributo CONTA, da tabela BANCO, só poderá receber valores de 1 a 99.999. Caso não seja obedecida essa regra, a ação será interrompida.

O exemplo abaixo garante que o atributo IDADE, da tabela PROFESSOR, não pode receber valores menores que 0.

```
1. FIREBIRD/ORACLE
2. ALTER TABLE PROFESSOR
3. ADD CONSTRAINT CHECK_IDADE
   CHECK(IDADE > 0)
```

As *check constraints* também podem ser definidas no momento da criação das tabelas, conforme o exemplo a seguir:

```
1. FIREBIRD
2. CREATE TABLE COMISSAO(
3. CODIGO SMALLINT NOT NULL,
4. NOME VARCHAR(50) NOT NULL,
5. VALOR NUMERIC(3,1)
     CHECK (VALOR > 0 AND VALOR <100),
6. CONSTRAINT COMISSAO_CODIGO_PK
     PRIMARY KEY(CODIGO))
```

```
1. ORACLE
2. CREATE TABLE COMISSAO(
3. CODIGO NUMBER(4) NOT NULL,
4. NOME VARCHAR2(50) NOT NULL,
5. VALOR NUMERIC(3,1)
     CHECK (VALOR > 0 AND VALOR <100),
6. CONSTRAINT COMISSAO_CODIGO_PK
     PRIMARY KEY(CODIGO))
```

A *constraint* do exemplo anterior garante que o atributo VALOR, da tabela COMISSAO, seja maior que 0 e menor que 100, garantindo que o valor da comissão seja armazenado de forma correta.

2.6 Restrição de Unicidade ou *Unique*

A restrição de unicidade diz que todos os dados contidos no atributo, ou no grupo de atributos, são únicos em relação a todas as outras linhas da tabela. Uma restrição de unicidade somente é violada se pelo menos duas linhas na tabela tenham os campos definidos como únicos com valores idênticos aos de uma outra linha. Note que, mesmo na presença da restrição de unicidade, um número ilimitado de linhas que contenham o valor **nulo** em pelo menos uma das colunas da restrição, está em conformidade com o padrão SQL.

Como de costume, a restrição de unicidade pode ser declarada a partir da criação das tabelas, ou posteriormente. A definição das mesmas a partir de uma *constraint* nomeada facilita a identificação.

O exemplo a seguir é semelhante à restrição de *primary key*.

```
1. FIREBIRD/ORACLE
2. ALTER TABLE INGRESSO
3. ADD CONSTRAINT UNIQUE_NUMERO
   UNIQUE(NUMERO)
```

O exemplo a seguir mostra a criação de uma regra de unicidade no ato da criação da tabela:

```
1. FIREBIRD
2. CREATE TABLE INGRESSO(
3. CODIGO INTEGER NOT NULL,
4. NUMERO INTEGER NOT NULL,
5. NOME VARCHAR(50) NOT NULL,
6. VALOR NUMERIC(9,3) NOT NULL,
7. DATA TIMESTAMP NOT NULL,
8. CONSTRAINT INGRESSO_CODIGO_PK
     PRIMARY KEY(CODIGO),
9. CONSTRAINT UNIQUE_NUMERO
     UNIQUE(NUMERO))
```

```
1. ORACLE
2. CREATE TABLE INGRESSO(
3. CODIGO NUMBER(6) NOT NULL,
4. NUMERO NUMBER(6) NOT NULL,
5. NOME VARCHAR2(50) NOT NULL,
6. VALOR NUMERIC(9,3) NOT NULL,
7. DATA DATE NOT NULL,
8. CONSTRAINT INGRESSO_CODIGO_PK
     PRIMARY KEY(CODIGO),
9. CONSTRAINT UNIQUE_NUMERO
     UNIQUE(NUMERO))
```

Outra forma de se criar uma restrição de unicidade é por meio do índice único. Este pode conter

vários atributos de uma determinada tabela, assegurando assim que aquele grupo de atributos não se repetirá, como demonstrado no exemplo abaixo:

```
1. FIREBIRD/ORACLE
2. CREATE UNIQUE INDEX
     UNIQUE_CODPRODUTO_CODVENDA
     ON ITEMVENDA(CODPRODUTO, CODVENDA)
```

No exemplo, foi criado o índice único chamado UNIQUE_CODPRODUTO_CODVENDA, que agrupa os atributos CODPRODUTO e CODVENDA, da tabela ITEMVENDA. Essa restrição garante que os produtos não serão repetidos para uma determinada venda.

2.7 *Stored Procedure*

Stored procedures são rotinas armazenadas em um determinado banco de dados. Elas permitem o acesso diretamente aos dados, sem qualquer intervenção do cliente. Além disso, não comprometem o tráfego da rede, pois são executadas diretamente no servidor de banco de dados. A *stored procedure* é um módulo independente de código, que pode ser executado durante ações de um determinado sistema, podendo retornar valores facilitando a implementação e ganhando, em desempenho, pois permite a divisão de tarefas complexas em módulos menores e mais lógicos.

É possível também que um procedimento chame outro, permitindo a construção de conjunto e de rotinas padronizadas com chamadas diferentes.

Na maioria das vezes, a grande utilidade desse método está em efetuar tarefas de processamento periódico e bastante complexo, tais como rotinas de fechamento do mês, processo de arquivamento de rotinas de cálculos, permitindo a entrada de parâmetros quando a tarefa é invocada, podendo retornar ou não valores para a aplicação que a chamou.

As *Stored Procedure* são criadas pelo comando CREATE PROCEDURE, o qual a sintaxe pode ser vista a seguir:

```
1. CREATE PROCEDURE NameProcedure
2. < parâmetros de entrada >
3. RETURNS
4. < parâmetros de saída >
5. AS
6. < declarações de variaveis locais >
7. BEGIN
8. /* COMANDOS DE PROCEDIMENTO */
9. END
```

Os parâmetros de entrada permitem à aplicação passar os valores que podem ser usados para modificar o comportamento da *stored procedure*. A **listagem 1** mostra o código de uma procedure que tem como objetivo calcular o total mensal da folha de pagamento de um determinado departamento.

O número do departamento será transmitido para *stored procedure,* como um parâmetro de entrada. A procedure devolve informações para a aplicação cliente através dos parâmetros de saída.

A procedure da **listagem 1** é chamada de TOTAL_MENSAL. Ela declara, como parâmetro de entrada, COD_DEPTO, do tipo INTEGER (FIREBIRD) e NUMBER (6) (ORACLE), e possui quatro parâmetros de saída: TOT_SALARIO, AVG_SALARIO, MIN_SALARIO e MAX_SALARIO, todos do tipo NUMERIC (9,3). Tanto os parâmetros de saída quanto os de entrada devem estar entre parênteses.

O comando SELECT é um *select* comum, que obtém as informações básicas que a *stored procedure* necessita. O comando SUSPEND pausa a *stored procedure* até que os clientes busquem os valores de saída.

É possível também declarar variáveis locais dentro da *stored procedure*. Essas variáveis só existem enquanto as *stored procedure* estão sendo executadas. As variáveis locais são declaradas depois da palavra-chave ***AS*** e antes da palavra chave ***BEGIN***, que indica o início do corpo da *procedure*.

A **listagem 2** utiliza variáveis e é baseada na estrutura da **figura 11**. A variável CODPESSOA recebe o valor resultado do *select* das tabelas

```
1. FIREBIRD
2. CREATE PROCEDURE TOTAL_MENSAL(
3. COD_DEPTO INTEGER)
4. RETURNS(
5. TOT_SALARIO NUMERIC(9,3),
6. AVG_SALARIO NUMERIC(9,3),
7. MIN_SALARIO NUMERIC(9,3),
8. MAX_SALARIO NUMERIC(9,3))
9. AS
10. BEGIN
11.   SELECT SUM(SALARIO), AVG(SALARIO),
   MIN(SALARIO),
12.     MAX(SALARIO)
13.   FROM DEPTO
14.   WHERE COD_DEPTO = :COD_DEPTO
15.   INTO :TOT_SALARIO, :AVG_SALARIO, :
   MIN_SALARIO,
16.     :MAX_SALARIO;
17. END


1. ORACLE
2. CREATE PROCEDURE TOTAL_MENSAL(
3. COD_DEPTO NUMBER(6))
4. RETURNS(
5. TOT_SALARIO NUMERIC(9,3),
6. AVG_SALARIO NUMERIC(9,3),
7. MIN_SALARIO NUMERIC(9,3),
8. MAX_SALARIO NUMERIC(9,3))
9. AS
10. BEGIN
11.   SELECT SUM(SALARIO), AVG(SALARIO),
   MIN(SALARIO),
12.     MAX(SALARIO)
13.   FROM DEPTO
14.   WHERE COD_DEPTO = :COD_DEPTO
15.   INTO :TOT_SALARIO, :AVG_SALARIO, :
   MIN_SALARIO,
16.     :MAX_SALARIO;
17. SUSPEND;
18. END
```

Listagem 1. Exemplo de stored procedure

CLIENTES E PESSOA. O valor recebido é o código da pessoa no BD, e é aproveitado na localização do vendedor, na qual ocorre um novo *select* na tabela de VENDEDOR. Nesse exemplo, a *stored procedure* foi criada para localizar e buscar os dados da tabela CLIENTES e VENDEDOR, com o mesmo código, ou seja, apenas para uma

Figura 11: DER stored procedure.

```
1. FIREBIRD
2. CREATE PROCEDURE DADOS_CLIENTE_VENDEDOR(
3.   COD_CLIENTE_VENDEDOR INTEGER)
4. RETURNS(
5.   CODCLIENTE INTEGER,
6.   NOMECLIENTE VARCHAR(60),
7.   CODVENDEDOR INTEGER,
8.   NOMEVENDEDOR VARCHAR(60),
9.   CNPJCLI CHAR(12),
10.  INSCESTADUALCLI CHAR(14),
11.  CNPJVEND CHAR(12),
12.  INSCESTADUALVEND CHAR(14))
13. AS
14. DECLARE VARIABLE CODPESSOA;
15. BEGIN
16.    SELECT C.CODIGO, P.NOME, P.CNPJ,
17.      P.INSCESTADUAL, P.CODIGO
18.    FROM CLIENTES C
19.      JOIN PESSOA P ON P.CODIGO=C.
    CODPESSOA
20.    WHERE C.CODIGO = :COD_CLIENTE_VENDEDOR
21.    INTO :CODCLIENTE, :NOMECLIENTE, :
    CNPJCLI,
22.    :INSCESTADUALCLI, :CODPESSOA;
23.
24. SELECT V.CODIGO, P.NOME, P.CNPJ,
25.    P.INSCESTADUAL
26. FROM VENDEDOR
27. WHERE V.CODPESSOA = :CODPESSOA
28. INTO :CODVENDEDOR, :NOMEVENDEDOR, :
    CNPJVEND,
29.    INSCESTADUALVEND;
30. END
```

```
1. ORACLE
2. CREATE PROCEDURE DADOS_CLIENTE(
3.   COD_CLIENTE_VENDEDOR NUMBER(6))
4. RETURNS(
5.   CODCLIENTE NUMBER(6),
6.   NOMECLIENTE VARCHAR2(60),
7.   CODVENDEDOR NUMBER(6),
8.   NOMEVENDEDOR VARCHAR2(60),
9.   CNPJCLI CHAR(12),
10.  INSCESTADUALCLI CHAR(14),
11.  CNPJVEND CHAR(12),
12.  INSCESTADUALVEND CHAR(14))
13. AS
14. _DECLARE VARIABLE CODPESSOA;
15. BEGIN
16.    SELECT C.CODIGO, P.NOME, P.CNPJ,
17.      P.INSCESTADUAL, P.CODIGO
18.    FROM CLIENTE C
19.      JOIN PESSOA P ON P.CODIGO=C.
    CODPESSOA
20.    WHERE C.CODIGO = :COD_CLIENTE_VENDEDOR
21.    INTO :CODCLIENTE, :NOMECLIENTE, :
    CNPJCLI,
22.    :INSCESTADUALCLI, :CODPESSOA;
23.
24. SELECT V.CODIGO, P.NOME, P.CNPJ,
25.    P.INSCESTADUAL
26. FROM VENDEDOR
27. WHERE V.CODPESSOA = :CODPESSOA
28. INTO :CODVENDEDOR, :NOMEVENDEDOR, :
    CNPJVEND,
29.    INSCESTADUALVEND;
30. SUSPEND;
31. END
```

Listagem 2. Criação de Stored Procedures

consulta rápida.

Alguns comandos da linguagem utilizada nas *Stored procedures* e *Triggers* lembram o Pascal como, por exemplo, o IF-THEN-ELSE ou WHILE.

As *stored procedures* dos SGBDs, de grande e médio porte, suportam a utilização dos comandos SELECT, INSERT, UPDATE e DELETE. Para todos os comandos, podem ser utilizadas variáveis locais ou parâmetros em qualquer lugar que o valor for aceito, como mostra a **listagem 3**.

O exemplo utiliza os comandos UPDATE, INSERT e DELETE dentro de uma stored procedure.

O Firebird permite a criação de ***select stored procedures***, que podem ser utilizadas em instruções ***select*** no lugar da tabela fonte de dados.

Com base nos exemplos e especificações, podemos tratar as *stored procedures* como um ótimo recurso para a aplicação da integridade, pois elas conseguem agilizar alguns processos, fazendo

```
1. FIREBIRD
2. CREATE PROCEDURE AJUSTA_CIDADE
3. AS
4. BEGIN
5.   UPDATE CIDADE SET CODUF='PR'
       WHERE NOME='LONDRINA';
6.   UPDATE CIDADE SET CODUF='SP'
       WHERE NOME='ASSIS';
7.   INSERT INTO CIDADE(CODIGO,
       NOME, CODUF)
       VALUES(100, 'BANDEIRANTES', 'PR');
8.   DELETE FROM CIDADE_VELHA;
9. END

1. ORACLE
2. CREATE PROCEDURE AJUSTA_CIDADE
3. AS
4. BEGIN
5.   UPDATE CIDADE SET CODUF="PR"
       WHERE NOME="LONDRINA";
6.   UPDATE CIDADE SET CODUF="SP"
       WHERE NOME="ASSIS";
7.   INSERT INTO CIDADE (CODIGO,
      NOME, CODUF)
      VALUES (100, "BANDEIRANTES", "PR");
8.   DELETE FROM CIDADE_VELHA;
9. END
```

Listagem 3. Procedure usando comandos de DML

com que a aplicação ganhe em desempenho, fator importantíssimo para médias e grandes aplicações.

2.8 Gatilhos ou *Triggers*

Os gatilhos ou *triggers* utilizam praticamente a mesma linguagem das stored procedures, mas possuem algumas diferenças:

- Não possuem parâmetros de entrada ou de saída;
- São disparados automaticamente devido a algum evento (inserção, atualização, remoção) ocorrido na tabela a qual o trigger está associado;

Podem auxiliar, e muito, na manutenção da integridade dos dados, pois são compartilhados por todas as aplicações clientes que acessam o banco de dados, pois estão codificados dentro do banco de dados, evitando a programação da lógica dentro de cada aplicação cliente. A redução no tráfego na rede também é grande, pois as operações são realizadas dentro do próprio servidor.

O trigger pode fazer a validação de dados e, caso encontre alguma inconsistência, poderá interromper o processo ao qual está associado (inserção, alteração, remoção), abortando a operação.

Abaixo temos um exemplo da sintaxe para a criação de um trigger:

```
1. CREATE TRIGGER name FOR table
2. ACTIVE BEFORE INSERT
3. AS
4. DECLARE VARIABLE variable tipo
5. BEGIN
6. /* BLOCO DE PROCEDIMENTO */
7. END
```

name: nome do *trigger*.

table: nome da tabela ou *view* a qual deseja ser criada o *trigger*.

ACTIVE/NACTIVE: opcional, especifica se o *trigger* está ativo ou inativo.

BEFORE UPDATE: especifica se o *trigger* dispara antes de uma atualização.

AFTER UPADATE: especifica se o *trigger* dispara após uma atualização.

BEFORE INSERT: especifica se o *trigger* dispara antes de uma inserção.

AFTER INSERT: especifica se o *trigger* dispara após uma inserção.

BEFORE DELETE: especifica se o *trigger* dispara antes de uma exclusão.

AFTER DELETE: especifica se o *trigger* dispara após uma exclusão.

DECLARE VARIABLE: declaração da variável que será usada no *trigger*.

Note que a execução de um trigger pode ocasionar a manipulação de dados em outras tabelas, criando um efeito cascata, por exemplo: se ocorrer uma alteração na TABELA A, pode-se disparar um *trigger* que atualize a TABELA B. A TABELA B, por sua vez, pode disparar um *trigger* que insere um novo valor na TABELA C, o que pode provocar um disparo de um *trigger* que atualiza a TABELA D, e assim sucessivamente.

Outro ponto importante que deve ser levado em conta é que o *trigger* faz parte da transação que o disparou, portanto um *commit* ou *rollback* aceita ou rejeita todas as operações desencadeadas pelo trigger.

Os eventos aos quais os triggers podem estar associados são:

```
1. BEFORE UPDATE
2. BEFORE DELETE
3. BEFORE INSERT
4. AFTER UPDATE
5. AFTER DELETE
6. AFTER INSERT
```

Triggers BEFORE são executados antes que a operação associada tenha sido executada. Triggers AFTER são executados depois da operação ter sido realizada. Podemos abortar a execução de um trigger ou operação disparando uma EXCEPTION.

A **listagem 4** mostra a criação de uma tabela FUNCIONARIO e uma *exception* FUNCIONARIO_SALARIO que será utilizada nos triggers.

```
1. FIREBIRD
2. CREATE TABLE FUNCIONARIO(
3. CODIGO INTEGER NOT NULL,
4. NOME VARCHAR(60) NOT NULL,
5. SALARIO NUMERIC(9,3)NOT NULL,
6. CONSTRAINT FUNCIONARIO_CODIGO_PK
   PRIMARY KEY(CODIGO));

1. CREATE EXCEPTION FUNCIONARIO_SALARIO
2. 'O salário do funcionário deve ser
   maior que 0';
3.
4. CREATE TRIGGER FUNCIONARIO_SALARIO FOR
   FUNCIONARIO
5. ACTIVE BEFORE INSERT POSITION 0
6. AS
7. BEGIN
8. IF (NEW.SALARIO <= 0 OR
       NEW.SALARIO IS NULL) THEN
9.   EXCEPTION FUNCIONARIO_SALARIO;
10. END

1. ORACLE
2. CREATE TABLE FUNCIONARIO(
3. CODIGO NUMBER(6) NOT NULL,
4. NOME VARCHAR2(60) NOT NULL,
5. SALARIO NUMBER(9,3)NOT NULL,
6. CONSTRAINT FUNCIONARIO_CODIGO_PK
   PRIMARY KEY(CODIGO))
7.
8. CREATE EXCEPTION FUNCIONARIO_SALARIO
9. 'O salário do funcionário deve ser
   maior que 0'
10.
11.
12. CREATE TRIGGER FUNCIONARIO_SALARIO FOR
    FUNCIONARIO
13. ACTIVE BEFORE INSERT POSITION 0
14. AS
15. BEGIN
16. IF (NEW.SALARIO <= 0 OR
        NEW.SALARIO IS NULL) THEN
17.   EXCEPTION FUNCIONARIO_SALARIO;
18. END
```

Listagem 4. Exemplo de utilização de triggers e exceções

No exemplo, o trigger impede que um novo funcionário seja cadastrado com um valor de salário menor ou igual a zero (ou nulo), disparando uma exceção.

No DER mostrado na **figura 12**, podemos identificar as tabelas NOTAFISCAL, NOTAFISCALITEM e DOCUMENTOFISCAL, sendo que NOTAFISCAL se relaciona com NOTAFISCALITEM. DOCUMENTOFISCAL é uma estrutura à aparte. Toda a vez que ocorrer um movimento na estrutura NOTAFISCALITEM, mais precisamente no atributo QTDE, tanto na inserção ou exclusão, é obrigatório que ocorra a mesma ação na estrutura de DOCUMENTOFISCAL, no atributo QTDE, para que possa ser fechado no final do mês e entregue à receita o valor dos movimentos para um determinado produto. Para solucionar esse problema, é criado um *trigger* para atualizar o atributo QTDE, da estrutura DOCUMENTOFISCAL.

A **listagem 5** mostra o código dos triggers fazendo uso das variáveis OLD e NEW, que fornecem respectivamente o valor antigo como o valor novo de qualquer atributo, para atualizar o respectivo registro na tabela DOCUMENTOFISCAL, no ato de uma inserção ou remoção de um registro na tabela NOTAFISCALITEM.

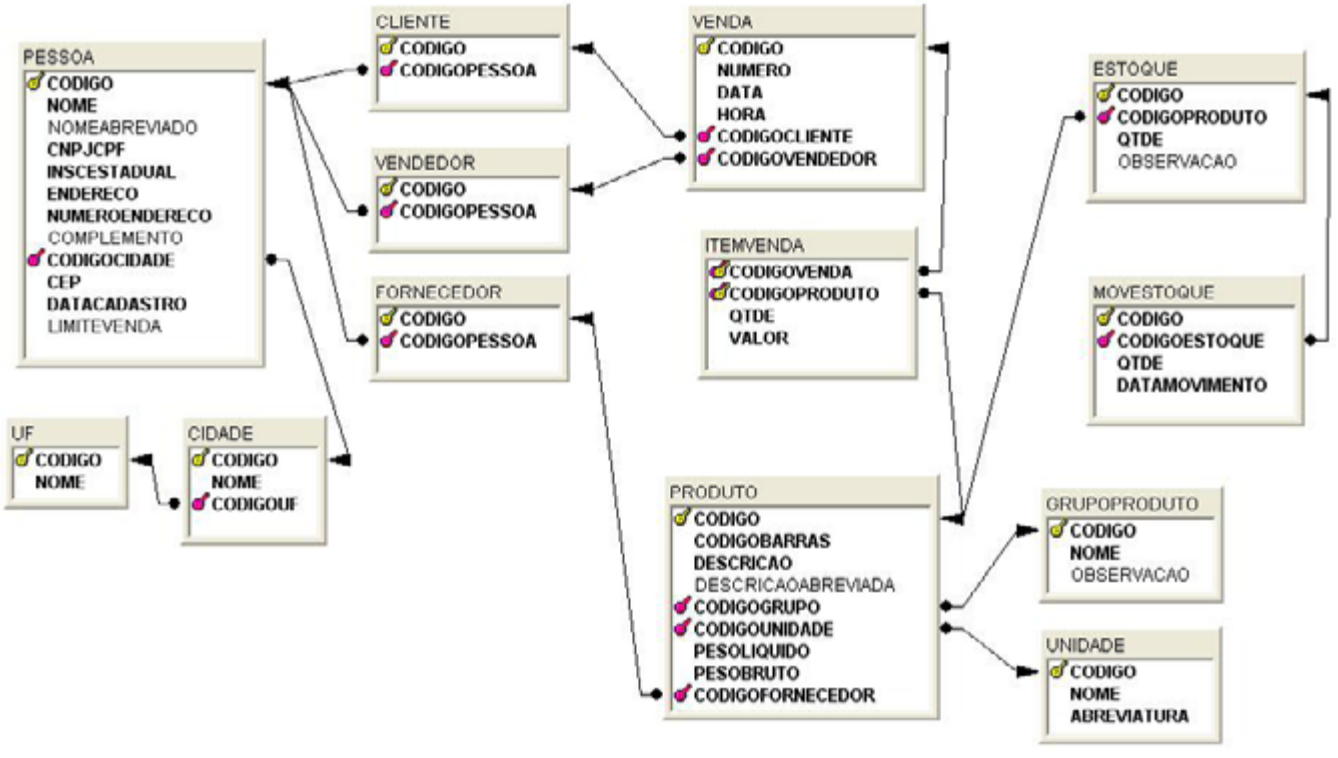


Figura 13: DER normalizado.

Figura 12: DER *trigger*

3. Exemplos práticos em PL/SQL.

A Linguagem PL/SQL (Procedural Language Extension to SQL) combina a facilidade de manipulação dos dados da linguagem SQL, com as facilidades de programação de uma linguagem procedural.

A **figura 13** mostra o DER dos módulos de venda e estoque de um sistema já existente. Os exemplos a seguir serão baseados nessa estrutura.

```
1. FIREBIRD

2. CREATE TRIGGER NOTAFISCALITEM _INSERIR
   FOR NOTAFISCALITEM
3. AFTER INSERT AS
4. BEGIN
5.   UPDATE DOCUMENTOFISCAL SET
            QTDE = QTDE + NEW.QTDE
6.   WHERE CODIGOPRODUTO = NEW.
   CODIGOPRODUTO;
7. END
8.
9.
10. CREATE TRIGGER NOTAFISCALITEM _EXCLUIR
   FOR NOTAFISCALITEM
11. AFTER DELETE
   AS
12. BEGIN
13.   UPDATE DOCUMENTOFISCAL SET
            QTDE = QTDE - OLD.QTDE
14.   WHERE CODIGOPRODUTO =
           OLD.CODIGOPRODUTO;
15. END



1. ORACLE

2. CREATE TRIGGER NOTAFISCALITEM_INSERIR
   FOR NOTAFISCALITEM
3. AFTER INSERT
   AS
4. BEGIN
5.   UPDATE DOCUMENTOFISCAL SET
            QTDE = QTDE + NEW.QTDE
6.   WHERE CODIGOPRODUTO =
           NEW.CODIGOPRODUTO;
7. END
8.
9. CREATE TRIGGER NOTAFISCALITEM _EXCLUIR
   FOR NOTAFISCALITEM
10. AFTER DELETE AS
11. BEGIN
12.   UPDATE DOCUMENTOFISCAL SET
            QTDE = QTDE - OLD.QTDE
13.   WHERE CODIGOPRODUTO =
           OLD.CODIGOPRODUTO;
14. END
```

Listagem 5. Triggers utilizando NEW e OLD

3.1 Criação das Estruturas

No processo de criação das estruturas, será aplicada todas as formas de garantir a restrição de integridade. O código abaixo cria duas tabelas simples, empregando somente as restrições básicas:

```
1. FIREBIRD

2. CREATE TABLE UNIDADE(
3. CODIGO INTEGER NOT NULL,
4. NOME VARCHAR(30) NOT NULL,
5. ABREVIATURA CHAR(3) NOT NULL,
6. CONSTRAINT UNIDADE_CODIGO_PK PRIMARY
   KEY(CODIGO));
7.
8. CREATE TABLE GRUPOPRODUTO(
9. CODIGO INTEGER NOT NULL,
10. NOME VARCHAR(30) NOT NULL,
11. OBSERVACAO VARCHAR(100),
12. CONSTRAINT GRUPOPRODUTO_CODIGO_PK
   PRIMARY KEY(CODIGO))

1. ORACLE

2. CREATE TABLE UNIDADE(
3. CODIGO NUMBER(5) NOT NULL,
4. NOME VARCHAR2(30) NOT NULL,
5. ABREVIATURA CHAR(3) NOT NULL,
6. CONSTRAINT UNIDADE_CODIGO_PK PRIMARY
   KEY(CODIGO));
7.
8. CREATE TABLE GRUPOPRODUTO(
9. CODIGO NUMBER(5) NOT NULL,
10. NOME VARCHAR2(30) NOT NULL,
11. OBSERVACAO VARCHAR2(100),
12. CONSTRAINT GRUPOPRODUTO_CODIGO_PK
   PRIMARY KEY(CODIGO))
```

Podemos ver que nas tabelas UNIDADE e GRUPOPRODUTO foram aplicadas restrições básicas como a definição de chave primária e campos não nulos. Começamos com essas tabelas por não terem ligação ou dependência com nenhuma outra tabela.

O código abaixo crias as tabelas UF e CIDADE, onde UF depende de CIDADE através da criação de uma chave estrangeira no campo CODIGOUF.

```
1. FIREBIRD

2. CREATE TABLE UF(
3. CODIGO CHAR(2) NOT NULL,
4. NOME VARCHAR(40) NOT NULL,
5. CONSTRAINT UF_CODIGO_PK PRIMARY
   KEY(CODIGO));
6.
7. CREATE TABLE CIDADE(
8. CODIGO INTEGER NOT NULL,
9. NOME VARCHAR(60) NOT NULL,
10. CODIGOUF CHAR(2) NOT NULL,
11. CONSTRAINT CIDADE_CODIGO_PK PRIMARY
   KEY(CODIGO),
12. CONSTRAINT CIDADE_CODIGOUF_FK FOREIGN
   KEY(CODIGOUF) REFERENCES UF(CODIGO))

1. ORACLE

2. CREATE TABLE UF(
3. CODIGO CHAR(2) NOT NULL,
4. NOME VARCHAR2(40) NOT NULL,
5. CONSTRAINT UF_CODIGO_PK PRIMARY
   KEY(CODIGO));
6.
7. CREATE TABLE CIDADE(
8. CODIGO NUMBER NOT NULL,
9. NOME VARCHAR2(60) NOT NULL,
10. CODIGOUF CHAR(2) NOT NULL,
11. CONSTRAINT CIDADE_CODIGO_PK PRIMARY
   KEY(CODIGO),
12. CONSTRAINT CIDADE_CODIGOUF_FK FOREIGN
   KEY(CODIGOUF) REFERENCES UF(CODIGO))
```

A **listagem 6** define a criação da tabela PESSOA, onde é criada uma integridade referencial pela *constraint* PESSOA_CODIGOCIDADE_FK que referencia a tabela PESSOA com a tabela CIDADE pelo atributo CODIGOCIDADE. A restrição *check* foi aplicada no atributo LIMITEVENDA que verifica se o atributo recebeu valores menores ou iguais a 0, se isso acontecer a restrição *check* não deixa o valor ser inserido.

```
1. FIREBIRD
2. CREATE TABLE PESSOA(
3. CODIGO INTEGER NOT NULL,
4. NOME VARCHAR(80) NOT NULL,
5. NOMEABREVIADO VARCHAR(40),
6. CNPJCPF CHAR(14) NOT NULL,
7. INSCESTADUAL CHAR(14) NOT NULL,
8. ENDERECO VARCHAR(60) NOT NULL,
9. NUMEROENDERECO INTEGER NOT NULL,
10. COMPLEMENTO VARCHAR(40),
11. CODIGOCIDADE INTEGER NOT NULL,
12. CEP CHAR(8) NOT NULL,
13. DATACADASTRO TIMESTAMP NOT NULL,
14. LIMITEVENDA NUMERIC(9,3)
    CHECK(LIMITEVENDA >= 0),
15. CONSTRAINT PESSOA_CODIGO_PK PRIMARY
    KEY(CODIGO),
16. CONSTRAINT PESSOA_CODIGOCIDADE_FK
    FOREIGN KEY(CODIGOCIDADE) REFERENCES
    CIDADE(CODIGO))

1. ORACLE
2. CREATE TABLE PESSOA(
3. CODIGO NUMBER(5) NOT NULL,
4. NOME VARCHAR2(80) NOT NULL,
5. NOMEABREVIADO VARCHAR2(40),
6. CNPJCPF CHAR(14) NOT NULL,
7. INSCESTADUAL CHAR(14) NOT NULL,
8. ENDERECO VARCHAR2(60) NOT NULL,
9. NUMEROENDERECO NUMBER(5) NOT NULL,
10. COMPLEMENTO VARCHAR2(40),
11. CODIGOCIDADE NUMBER(5) NOT NULL,
12. CEP CHAR(8) NOT NULL,
13. DATACADASTRO DATE NOT NULL,
14. LIMITEVENDA NUMERIC(9,3)
    CHECK(LIMITEVENDA >= 0),
15. CONSTRAINT PESSOA_CODIGO_PK PRIMARY
    KEY(CODIGO),
16. CONSTRAINT PESSOA_CODIGOCIDADE_FK
    FOREIGN KEY(CODIGOCIDADE) REFERENCES
    CIDADE(CODIGO))
```

Listagem 6. Criação da tabela PESSOA

A **listagem 7** define a criação das demais tabelas e suas respectivas regras de integridade e relacionamento.

```
1. FIREBIRD
2. CREATE TABLE CLIENTE(
3. CODIGO INTEGER NOT NULL,
4. CODIGOPESSOA INTEGER NOT NULL,
5. CONSTRAINT CLIENTE_CODIGO_PK PRIMARY
   KEY(CODIGO),
6. CONSTRAINT CLIENTE_CODIGOPESSOA_FK
   FOREIGN KEY(CODIGOPESSOA) REFERENCES
   PESSOA(CODIGO));
7.
8. CREATE TABLE FORNECEDOR(
9. CODIGO INTEGER NOT NULL,
10. CODIGOPESSOA INTEGER NOT NULL,
11. CONSTRAINT FORNECEDOR_CODIGO_PK
    PRIMARY KEY(CODIGO),
12. CONSTRAINT FORNECEDOR_CODIGOPESSOA_FK
    FOREIGN KEY(CODIGOPESSOA) REFERENCES
    PESSOA(CODIGO));
13.
14. CREATE TABLE VENDEDOR(
15. CODIGO INTEGER NOT NULL,
16. CODIGOPESSOA INTEGER NOT NULL,
17. CONSTRAINT VENDEDOR_CODIGO_PK PRIMARY
    KEY(CODIGO),
18. CONSTRAINT VENDEDOR_CODIGOPESSOA_FK
    FOREIGN KEY(CODIGOPESSOA) REFERENCES
    PESSOA(CODIGO));
19.
20. CREATE TABLE PRODUTO(
21. CODIGO INTEGER NOT NULL,
22. CODIGOBARRAS VARCHAR(24)NOT NULL,
23. DESCRICAO VARCHAR(60) NOT NULL,
24. DESCRICAOABREVIADA VARCHAR(40),
25. CODIGOGRUPO INTEGER NOT NULL,
26. CODIGOUNIDADE INTEGER NOT NULL,
27. PESOLIQUIDO NUMERIC(9,3) NOT NULL
    CHECK(PESOLIQUIDO > 0),
28. PESOBRUTO NUMERIC(9,3) NOT NULL
    CHECK(PESOBRUTO > 0),
29. CODIGOFORNECEDOR INTEGER NOT NULL,
30. CONSTRAINT PRODUTO_CODIGO_PK PRIMARY
    KEY(CODIGO),
31. CONSTRAINT PRODUTO_CODIGOGRUPO_FK
    FOREIGN KEY(CODIGOGRUPO) REFERENCES
    GRUPOPRODUTO(CODIGO),
32. CONSTRAINT PRODUTO_CODIGOUNIDADE_FK
    FOREIGN KEY(CODIGOUNIDADE) REFERENCES
    UNIDADE(CODIGO),
33. CONSTRAINT PRODUTO_CODIGOFORNECEDOR_
    FK FOREIGN KEY(CODIGOFORNECEDOR)
    REFERENCES FORNECEDOR(CODIGO));
34. CREATE TABLE ESTOQUE(
35. CODIGO INTEGER NOT NULL,
36. CODIGOPRODUTO INTEGER NOT NULL,
37. QTDE INTEGER NOT NULL,
38. OBSERVACAO VARCHAR(100),
39. CONSTRAINT ESTOQUE_CODIGO_PK PRIMARY
    KEY(CODIGO),
40. CONSTRAINT ESTOQUE_CODIGOPRODUTO_FK
    FOREIGN KEY(CODIGOPRODUTO) REFERENCES
    PRODUTO(CODIGO));
41.
42. CREATE TABLE MOVESTOQUE(
43. CODIGO INTEGER NOT NULL,
44. CODIGOESTOQUE INTEGER NOT NULL,
45. QTDE INTEGER NOT NULL,
46. DATAMOVIMENTO TIMESTAMP NOT NULL,
47. CONSTRAINT MOVESTOQUE_CODIGO_PK
    PRIMARY KEY(CODIGO),
48. CONSTRAINT MOVESTOQUE_CODIGOESTOQUE_FK
      FOREIGN KEY(CODIGOESTOQUE)
      REFERENCES ESTOQUE(CODIGO));
49.
50. CREATE TABLE VENDA(
51. CODIGO INTEGER NOT NULL,
52. NUMERO INTEGER NOT NULL
          CHECK NUMERO > 0,
53. DATA TIMESTAMP NOT NULL,
54. HORA TIMESTAMP NOT NULL,
55. CODIGOCLIENTE INTEGER NOT NULL,
56. CODIGOVENDEDOR INTEGER NOT NULL,
57. CONSTRAINT VENDA_CODIGO_PK
       PRIMARY KEY(CODIGO),
58. CONSTRAINT VENDA_CODIGOCLIENTE_FK
       FOREIGN KEY(CODIGOCLIENTE)
       REFERENCES CLIENTE(CODIGO),
59. CONSTRAINT VENDA_CODIGOVENDEDOR_FK
       FOREIGN KEY(CODIGOVENDEDOR)
       REFERENCES VENDEDOR(CODIGO),
60. CONSTRAINT VENDA_NUMERO_UIDX
    UNIQUE(NUMERO));
61.
62. CREATE TABLE ITEMVENDA(
63. CODIGOVENDA INTEGER NOT NULL,
64. CODIGOPRODUTO INTEGER NOT NULL,
65. QTDE INTEGER NOT NULL
         CHECK(QTDE > 0),
66. VALOR NUMERIC(9,3) NOT NULL
          CHECK(VALOR > 0),
67. CONSTRAINT ITEMVENDA_CODPRODCODVENDA_
```

```
   PK PRIMARY KEY(CODIGOVENDA,
   CODIGOPRODUTO),
68. CONSTRAINT ITEMVENDA_CODIGOVENDA_FK
   FOREIGN KEY(CODIGOVENDA)
           REFERENCES VENDA(CODIGO),
69. CONSTRAINT ITEMVENDA_CODIGOPRODUTO_FK
   FOREIGN KEY(CODIGOPRODUTO)
           REFERENCES PRODUTO(CODIGO));
```

```
1. ORACLE

2. CREATE TABLE CLIENTE(
3. CODIGO NUMBER(5) NOT NULL,
4. CODIGOPESSOA NUMBER(5) NOT NULL,
5. CONSTRAINT CLIENTE_CODIGO_PK PRIMARY
   KEY(CODIGO),
6. CONSTRAINT CLIENTE_CODIGOPESSOA_FK
   FOREIGN KEY(CODIGOPESSOA)
           REFERENCES PESSOA(CODIGO));
7.
8. CREATE TABLE FORNECEDOR(
9. CODIGO NUMBER(5) NOT NULL,
10. CODIGOPESSOA NUMBER(5) NOT NULL,
11. CONSTRAINT FORNECEDOR_CODIGO_PK
   PRIMARY KEY(CODIGO),
12. CONSTRAINT FORNECEDOR_CODIGOPESSOA_FK
   FOREIGN KEY(CODIGOPESSOA)
           REFERENCES PESSOA(CODIGO));
13.
14. CREATE TABLE VENDEDOR(
15. CODIGO NUMBER(5) NOT NULL,
16. CODIGOPESSOA NUMBER(5) NOT NULL,
17. CONSTRAINT VENDEDOR_CODIGO_PK
              PRIMARY KEY(CODIGO),
18. CONSTRAINT VENDEDOR_CODIGOPESSOA_FK
   FOREIGN KEY(CODIGOPESSOA)
           REFERENCES PESSOA(CODIGO));
```

```
1. CREATE TABLE PRODUTO(
2. CODIGO NUMBER(5) NOT NULL,
3. CODIGOBARRAS VARCHAR2(24)NOT NULL,
4. DESCRICAO VARCHAR2(60) NOT NULL,
5. DESCRICAOABREVIADA VARCHAR2(40),
6. CODIGOGRUPO NUMBER(5) NOT NULL,
7. CODIGOUNIDADE NUMBER(5) NOT NULL,
8. PESOLIQUIDO NUMERIC(9,3) NOT NULL
   CHECK(PESOLIQUIDO > 0),
9. PESOBRUTO NUMERIC(9,3) NOT NULL
   CHECK(PESOBRUTO > 0),
10. CODIGOFORNECEDOR NUMBER(5) NOT NULL,
11. CONSTRAINT PRODUTO_CODIGO_PK PRIMARY
   KEY(CODIGO),
12. CONSTRAINT PRODUTO_CODIGOGRUPO_FK
   FOREIGN KEY(CODIGOGRUPO) REFERENCES
   GRUPOPRODUTO(CODIGO),
13. CONSTRAINT PRODUTO_CODIGOUNIDADE_FK
   FOREIGN KEY(CODIGOUNIDADE) REFERENCES
   UNIDADE(CODIGO),
14. CONSTRAINT PRODUTO_CODIGOFORNECEDOR_
   FK FOREIGN KEY(CODIGOFORNECEDOR)
   REFERENCES FORNECEDOR(CODIGO));
15.
16. CREATE TABLE ESTOQUE(
17. CODIGO NUMBER(5) NOT NULL,
18. CODIGOPRODUTO NUMBER NOT NULL,
19. QTDE NUMBER NOT NULL,
20. OBSERVACAO VARCHAR2(100),
21. CONSTRAINT ESTOQUE_CODIGO_PK PRIMARY
   KEY(CODIGO),
22. CONSTRAINT ESTOQUE_CODIGOPRODUTO_FK
   FOREIGN KEY(CODIGOPRODUTO) REFERENCES
   PRODUTO(CODIGO));
23.
24. CREATE TABLE MOVESTOQUE(
25. CODIGO NUMBER(5) NOT NULL,
26. CODIGOESTOQUE NUMBER(5) NOT NULL,
27. QTDE NUMBER(5) NOT NULL,
28. DATAMOVIMENTO DATE NOT NULL,
29. CONSTRAINT MOVESTOQUE_CODIGO_PK
   PRIMARY KEY(CODIGO),
30. CONSTRAINT MOVESTOQUE_CODIGOESTOQUE_FK
   FOREIGN KEY(CODIGOESTOQUE) REFERENCES
   ESTOQUE(CODIGO));
31.
32. CREATE TABLE VENDA(
33. CODIGO NUMBER(5) NOT NULL,
34. NUMERO NUMBER(6) NOT NULL
          CHECK NUMERO > 0,
35. DATA DATE NOT NULL,
36. HORA TIME NOT NULL,
37. CODIGOCLIENTE NUMBER(5) NOT NULL,
38. CODIGOVENDEDOR NUMBER(5) NOT NULL,
39. CONSTRAINT VENDA_CODIGO_PK PRIMARY
   KEY(CODIGO),
40. CONSTRAINT VENDA_CODIGOCLIENTE_FK
   FOREIGN KEY(CODIGOCLIENTE) REFERENCES
   CLIENTE(CODIGO),
41. CONSTRAINT VENDA_CODIGOVENDEDOR_FK
   FOREIGN KEY(CODIGOVENDEDOR) REFERENCES
   VENDEDOR(CODIGO),
42. CONSTRAINT VENDA_NUMERO_UIDX
   UNIQUE(NUMERO));
43.
44. CREATE TABLE ITEMVENDA(
45. CODIGOVENDA NUMBER(5) NOT NULL,
46. CODIGOPRODUTO NUMBER(5) NOT NULL,
47. QTDE NUMBER(5) NOT NULL
        CHECK(QTDE > 0),
48. VALOR NUMERIC(9,3) NOT NULL
        CHECK(VALOR > 0),
49. CONSTRAINT ITEMVENDA_CODPRODCODVENDA_
   PK
        PRIMARY KEY(CODIGOVENDA,
   CODIGOPRODUTO),
50. CONSTRAINT ITEMVENDA_CODIGOVENDA_FK
     FOREIGN KEY(CODIGOVENDA)
     REFERENCES VENDA(CODIGO),
51. CONSTRAINT ITEMVENDA_CODIGOPRODUTO_FK
   FOREIGN KEY(CODIGOPRODUTO) REFERENCES
   PRODUTO(CODIGO));
```

Listagem 7. Criação das demais tabelas

3.2 Criação das *Triggers* e *Stored Procedures*

A **listagem 8** define a criação da *procedure* VENDAPRODUTO, que tem como finalidade listar os produtos vendidos em um determinado período. A *procedure* é criada baseada nas tabelas VENDA e ITEMVENDA, tendo como parâmetros de entrada a data inicial e data final que definem o período que será pesquisado. Serão retornados da *procedure* os códigos dos produtos, as quantidades e os valores vendidos dentro do período.

No corpo da *procedure* é feito um SELECT listando o campo CODIGOPRODUTO, a soma do campos QTDE, obtendo o valor total do item através da soma dos campos QTDE e VALOR.

A **listagem 9** mostra a criação de dois triggers. O *trigger* MOVESTOQUE_EXCLUIR é disparado após uma remoção de um registro na tabela MOVESTOQUE – que armazena a movimentação do estoque dos produtos. Sua função é atualizar o saldo do estoque na tabela ESTOQUE para o produto que foi removido. O trigger MOVESTOQUE_INSERIR age de forma semelhante, mas no momento de uma inclusão na tabela MOVESTOQUE. Observe a utilização das variáveis NEW e OLD para obter o valor atual e anterior dos campos.

```
1. FIREBIRD
2. CREATE PROCEDURE VENDAPRODUTO(
3.    DATAINICIAL DATE,
4.    DATAFINAL DATE)
5. RETURNS(
6.    CODIGOPRODUTO INTEGER,
7.    QTDE INTEGER,
8.    VALORTOTAL NUMERIC(9,3))
9. AS
10.   BEGIN
11.     FOR
12.      SELECT I.CODIGOPRODUTO,
          SUM(I.QTDE) QTDE,
13.       SUM(I.QTDE * I.VALOR) VALORTOTAL
14.      FROM ITEMVENDA I
15.         JOIN VENDA V ON
                V.CODIGO = I.CODIGOVENDA
16.      WHERE (V.DATA>=:DATAINICIAL) AND
               (V.DATA<=DATAFINAL)
17.       GROUP BY I.CODIGOPRODUTO
18. INTO :CODIGOPRODUTO, :QTDE,
        :VALORTOTAL
19. DO SUSPEND;
20. END

1. ORACLE
2. CREATE PROCEDURE VENDAPRODUTO(
3.    DATAINICIAL DATE,
4.    DATAFINAL DATE)
5. RETURNS(
6.    CODIGOPRODUTO NUMBER(5),
7.    QTDE NUMBER(5),
8.    VALORTOTAL NUMERIC(9,3))
9. AS
10.   BEGIN
11.     FOR
12.      SELECT I.CODIGOPRODUTO,
          SUM(I.QTDE) QTDE,
13.       SUM(I.QTDE * I.VALOR) VALORTOTAL
14.      FROM ITEMVENDA I
15.         JOIN VENDA V ON
             V.CODIGO = I.CODIGOVENDA
16.      WHERE (V.DATA>=:DATAINICIAL) AND
               (V.DATA<=DATAFINAL)
17.      GROUP BY I.CODIGOPRODUTO
18. INTO :CODIGOPRODUTO, :QTDE,
        :VALORTOTAL
19. DO
20.   SUSPEND;
21. END
```

Listagem 8. Obtendo os produtos vendidos em um período

O código abaixo mostra o uso de exceções para interromper um processo em andamento, exemplo: Por definição, a tabela MOVESTOQUE não deve aceitar alteração nos seus dados. Para garantir que essa regra não seja violada, foi criado um trigger do tipo BEFORE UPDATE que impede qualquer tentativa de alteração dos dados na tabela.

```
1. FIREBIRD/ORACLE

2. CREATE EXCEPTION MOVESTOQUEALTERAR
   'A TABELA MOVESTOQUE NÃO PERMITE
   A ALTERAÇÃO, SOMENTE INSERÇÃO E
   EXCLUSÃO´;
3.
4. CREATE TRIGGER MOVESTOQUE_ALTERAR FOR
   MOVESTOQUE BEFORE UPDATE AS
5. BEGIN
6.   EXCEPTION MOVESTOQUEALTERAR;
7. END
```

Baseado em algumas restrições que foram impostas na criação da estrutura de dados, como a proibição da exclusão de uma venda já efetuada, a forma ideal de atender essa solicitação é criando uma exceção que será acionada pelo *trigger* quando a violação ocorrer, como mostrado no código abaixo:

```
1. FIREBIRD/ORACLE
2. CREATE EXCEPTION
   VENDAEXCLUIR 'A
   TABELA VENDA NÃO
   PERMITE EXCLUSÃO';
3.
4. CREATE TRIGGER
   VENDA_EXCLUIR FOR
   VENDA
5.   BEFORE DELETE AS
6.     BEGIN
7.       EXCEPTION
   VENDAEXCLUIR;
8.     END
```

```
1. FIREBIRD/ORACLE
2.
3. CREATE TRIGGER MOVESTOQUE_EXCLUIR FOR
   MOVESTOQUE
4. AFTER DELETE AS
5. BEGIN
6.   UPDATE ESTOQUE SET
            QTDE = QTDE - OLD.QTDE
7.   WHERE CODIGO = OLD.CODIGOESTOQUE;
8. END
9.
10. CREATE TRIGGER MOVESTOQUE_INSERIR FOR
    MOVESTOQUE
11. AFTER INSERT AS
12. BEGIN
13.   UPDATE ESTOQUE SET
            QTDE = QTDE + NEW.QTDE
14.   WHERE CODIGO = NEW.CODIGOESTOQUE;
15. END
```

Listagem 9. Triggers para atualização do estoque

CONSIDERAÇÕES FINAIS

Neste artigo, realizamos o estudo das formas de garantir restrição de integridade diretamente nos Bancos de Dados, nos quais essas formas são recursos que os Sistemas Gerenciadores de Banco de Dados (SGBD), de médio e grande porte, oferecem para o desenvolvimento das estruturas de dados.

Aplicando-se as formas de garantia de restrição de integridade nas estruturas, é possível garantir um alto grau de confiabilidade nas informações, evitando a inconsistência nos dados ou, até mesmo, facilitando futuras atualizações que uma estrutura confiável pode vir a oferecer.

REFERÊNCIAS BIBLIOGRÁFICAS

CORDEIRO, Robson L. F.; SANTOS, C. S.; GALANTE, R. M. **Classificação de restrição de integridade em Banco de Dados Temporal**. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 2005, Porto Alegre. Anais... Porto Alegre, 2005.

DANKER, Pablo Ricardo; ZIMMERMANN, Roberto J.; KUROSHI, Ricardo. **Integridade e desempenho em Banco de Dados.** In: CONGRESSO BANCO DE DADOS, 2, 2003, São José. Anais... São José, 2003.

DATE, C. J.. **Introdução a Sistemas de Banco de Dados**. Rio de Janeiro: Campus, 1990.

KADE, Adrovani Márquez. **Banco de Dados I**: restrições de integridade. Montevidéu: Universidade Regional Integrada do Alto Uruguai e das Missões, 2004.

KORTH, Henry F.; SILBERSCHATZ, Abraham. **Sistemas de Banco de Dados**. São Paulo: Makron Books, 1993.

MACHADO, Felipe Nery R.; ABREU, Maurício. **Projeto de Banco de Dados**. :Érica

POLETTO, Alex S. R. S. - **Apostilas de Banco de Dados e Ambiente de Desenvolvimento**

Autor:

Robert Nunes

Mini-curriculo

Experiência profissional em consultoria, análise e desenvolvimento de sistemas de informação, Possui conhecimento e familiaridade com as linguagens Java, Visual Basic, C++, C e especialista em Delphi. Utiliza banco de Dados Interbase, Firebird, MySql, SQLServer, Oracle e Access.

Email: robertcnunes@yahoo.com.br

A escolha do banco de dados

por Carlos H. Cantu

A escolha de um banco de dados por uma empresa é um passo extremamente importante e deve ser analisado detalhadamente, a fim de evitar surpresas nem sempre agradáveis quando o mesmo já estiver em utilização.

O banco de dados pode armazenar informações preciosas (e muitas vezes sigilosas) sobre a "vida" da empresa, tornando-se uma peça fundamental para o seu funcionamento. Um banco de dados parado gera problemas em todos os níveis, vejamos:

- Para o dono da empresa, significa perda de dinheiro, pois pedidos, notas fiscais e cobranças deixam de ser emitidas. Para aumentar ainda mais o desespero, relatórios gerenciais e análises ficam indisponíveis!
- Para o pessoal da área técnica (o famoso "pessoal da informática"), significa dor de cabeça, pressão, e muitas vezes viradas de noite trabalhando para colocar tudo no ar novamente.
- Para o usuário final dos sistemas que utilizam o banco de dados, geralmente o sentimento é de impotência, pois lhe falta sua principal ferramenta de trabalho.

Fica obvio que é necessário escolher um produto que não lhe deixe "na mão" em uma situação de emergência. Mas isso não depende somente do banco de dados, e sim de uma política mais abrangente, envolvendo backups rotineiros, manutenção nas máquinas (principalmente nos servidores), uso de no-breaks, sistemas RAID, links de rede redundantes e, obviamente, a criação de uma política de segurança, a fim de evitar que os dados caiam nas mãos das pessoas erradas.

Até pouco tempo atrás, apenas grandes SGBDs (Sistemas Gerenciadores de Banco de Dados) como, por exemplo, o *Oracle*, eram capazes de oferecer segurança e recursos sofisticados para a manipulação de dados.

Nos últimos anos, a história mudou rapidamente. Inúmeros bancos de dados gratuitos e *Open Source* surgiram ou se aprimoraram, e hoje ocupam uma fatia cada vez maior desse mercado. Com a popularização do **Linux**, principalmente em servidores, as empresas estão perdendo o medo e a desconfiança, e olhando de forma diferente os softwares Open Source.

Entre os bancos de dados **Open Source** mais difundidos atualmente, podemos citar o **MySQL** (muito usado em sites na internet), **Firebird** e **PostgreSQL**.

O crescimento dos bancos gratuitos e/ou open source tem provocado situações que até pouco tempo atrás seriam difíceis de imaginar: grandes empresas, como a Microsoft, IBM e a própria Oracle, estão disponibilizando versões gratuitas dos seus produtos. No entanto, vale lembrar que geralmente essas versões possuem algum tipo de limitação, e não tem código aberto, ou seja, você não tem acesso ao código fonte e não pode fazer alterações no mesmo. Sua finalidade é atrair novos usuários, que no futuro possivelmente terão que migrar para uma versão paga do produto.

Além do fator **preço**, onde os bancos open source/gratuitos são praticamente **imbatíveis**, outros fatores podem influenciar na escolha dos mesmos:

- **Multiplataforma**. Grande parte dos bancos open source rodam em vários sistemas operacionais, ficando o usuário livre para escolher o S.O. que mais lhe convém;
- **Possibilidade de adaptações**. A maioria dos bancos com código aberto possui licenças onde é permitido que cada usuário modifique ou personalize o produto para suas próprias necessidades, inclusive adicionando novos recursos;
- **Segurança**. Pelo fato de ter o código aberto, a detecção e correção de bugs ou falhas de segurança geralmente se dá bem mais rapidamente do que em um produto com código fechado;
- **Suporte**. É fácil encontrar comunidades "*on-line*" na internet, dedicadas ao suporte dos bancos de dados open source. Geralmente as dúvidas são respondidas nas listas ou fóruns de discussão em questão de horas ou mesmo minutos. A adoção cada vez maior dos BDs Open Source está provocando o aparecimento de empresas especializadas em consultoria para esses bancos, uma opção para aqueles que precisam da segurança de um suporte especializado ou contratado.

Mas o que devemos levar em conta?

Abaixo listo alguns pontos que devem ser levados em conta na hora de escolher um banco de dados.

1. **Sistemas** – Um banco de dados dificilmente "anda desacompanhado". Há sempre uma aplicação associada que fornece a interface com o usuário, para que ele possa manipular as informações armazenadas e extraí-las de forma adequada para serem analisadas. Sendo assim, os "sistemas" ou aplicações utilizadas determinam diretamente os bancos que poderão ou não entrar na "briga". Se a aplicação escolhida suporta o banco X ou Y, os bancos A,B e C já estão descartados.
2. **Preço** – fator sempre importante – e aqui os bancos gratuitos levam vantagem indiscutível.
3. **Sistema operacional** – Não adianta querer rodar o *MS SQL Server* no Linux, pois ele só roda no Windows! O sistema operacional adotado no servidor de banco de dados limita diretamente as opções de SGBDs, ou vice-versa.
4. **Suporte** – Ok, então você buscou no Google e descobriu um novo banco de

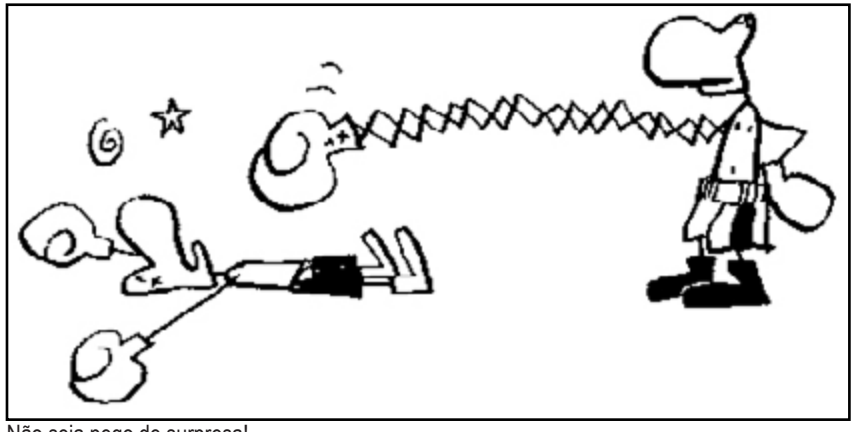
dados que parece ser uma maravilha! Mas e aí? Será que aquele grupo de 20 usuários listados no site são suficientes para lhe ajudar nos momentos difíceis? A existência de comunidades fortes e consolidadas, ou de empresas de consultoria especializadas é um fator importante, pois quando o "bicho pegar", você pode precisar de soluções rápidas para os seus problemas, e deve saber com quem pode contar..

5. **Hardware** – Não adianta querer rodar o Oracle no seu velho Pentium 233Mhz. Verifique os *requisitos mínimos* exigidos pelo SGBD e veja se eles se enquadram na sua realidade. Vale lembrar que **nem sempre o que está listado como requisito mínimo é sinônimo de boa performance**. Faça alguns testes e previna-se contra surpresas desagradáveis. Melhor perder tempo na escolha, do que depois ter que mudar de banco.
6. **Pessoal especializado** – Certo... então toda a sua equipe já é especializada em Oracle e você resolveu usar o PostgreSQL. *Que beleza!* Não precisa ném ir se consultar com um "pai de santo" para descobrir que problemas estão por vir. Se já existe pessoal especializado em determinada tecnologia, a mudança para outro SGBD exigirá um novo treinamento, a fim de que fiquem familiarizados com o mesmo, e possam dar conta de mantê-lo em perfeito funcionamento. Isso deve ser levado em conta antes de realizar qualquer mudança de SGBD.
7. **Recursos** – Antes de escolher um SGBD, devemos verificar se ele oferece todos os recursos que necessitamos, por exemplo: Sua aplicação precisa rodar em full-time (também chamado 24x7)? Então um banco de dados que exija acesso exclusivo para realização de backups, com certeza não é a melhor opção.

O que mais preciso saber?

É necessário ter consciência de que o SGBD é parte fundamental do processo, mas **não é o único ponto de preocupação**.

Não há SGBD que resista a um problema de hardware, ou ao ataque de um vírus fatal, da faxineira que resolve testar pra que serve aquele botãozinho na frente do micro, ou mesmo as mãos de um usuário leigo curioso. Se você não quer ficar na mão, alguns cuidados devem ser tomados. Eis aqui alguns deles:



Não seja pego de surpresa!

- Definição de uma política de segurança para a rede. É importante a utilização de um sistema operacional seguro e estável, principalmente no servidor! Linux ou Windows 200x são os mais indicados, lembrando que você deve escolher um sistema operacional suportado pelo SGBD. Lembre-se: o usuário comum não precisa ter acesso físico ao banco de dados! Crie níveis de segurança compatíveis com os papéis de cada usuário;
- Crie backups periodicamente e armazene-os em um local seguro. Em caso de "desastre" e quando nada mais adiantar, eles serão sua única opção e podem garantir o seu emprego;
- Se possível, utilize uma estrutura de RAID baseada em hardware e com tolerância a falhas. Isso é especialmente importante para SGBDs que devem rodar no esquema 24x7;
- Não se esqueça que um hardware confiável é a base para o bom funcionamento da sua estrutura de informática. Placas de rede "*chingling*", bem comos roteadores, switchers ou hubs de má qualidade são responsáveis geralmente por corrupção de dados, perda de pacotes, lentidão e outra infinidade de problemas que pode afetar desde um terminal isolado como toda a rede;
- Monitoramento constante é essencial para evitar ser pego de surpresa. Faça análise periódica do hardware envolvido, dos arquivos de log do servidor, das estatíticas dos bancos de dados, etc;

Especialize-se!

Como consultor de bancos de dados **Firebird**, já ouvi inúmeras vezes usuários reclamando que o SGBD corrompe com facilidade, ou que é muito lento. Na imensa maioria das vezes, o culpado não é o SGBD, e sim o desenvolvedor! Mostrarei alguns exemplos reais, que podem valer não só para o Firebird, como para outros SGBDs:

O banco de dados corrompe muito facilmente – Pois bem, será que milhões de pessoas em todo o mundo resolveram ser masoquistas e decidiram usar um banco de dados "tão" instável?

Acredito que não! Citando o Firebird como exemplo, ele é tão leve que pode ser instalado em um *Pentium I* e rodar com boa performance. Isso é um ponto positivo, mas pode criar problemas. Algumas pessoas acreditam que por esse fato, podem utilizar para servidor de BD aquele velho Pentium 233Mhz, com HD de 200MB, Windows 95 e 64MB de RAM. *Vai rodar?* Sim! *Vai ser confiável?* Claro que não! O HD já deve estar no final da sua vida útil, bem como outras partes do hardware. Na primeira falha, seu BD pode se comprometer. *Culpa do SGBD?* Obvio que não!

Os grandes bancos de dados, como Oracle, SQLServer, DB2, etc. são muito mais pesados, exigem hardware mais "parrudo" e consequentemente mais atual, sendo impossível utilizá-los em uma máquina como a descrita no exemplo acima. Só esse fato já obriga o desenvolvedor a escolher um hardware de capacidade e qualidade superior. Portanto, quando alguém disser que banco Open Source é muito instável ou dá muito problema, comece olhando onde o individuo instalou o SGBD.

O servidor é muito lento – Não me canso de "ouvir" isso nos diversos fóruns de discussão que participo. Mais uma vez, na sua imensa maioria, as reclamações são resultado da falta de especialização ou de conhecimento do desenvolvedor com o produto que está sendo utilizado. A pressa ou mesmo a preguiça, faz com que muitas pessoas nem sequer leiam as informações básicas, *sim! aquelas que estão disponíveis em 500 mil sites na internet, arquivos leiame, FAQs, etc*. Saem já programando, sem saber trabalhar adequadamente como a tecnologia escolhida. Muitas estão acostumadas com bancos de dados **desktop** (clipper, dbase, paradox, etc.); quando mudam para um banco **cliente/servidor**, não enxergam que a mudança não é somente de SGBD, mas sim de todo o conceito de programação envolvida. Abrir um "browse" no clipper em uma tabela com milhões de registros é muito fácil e rápido. Faça isso em um banco cliente/servidor e terá a nítida impressão que a mudança de tecnologia foi um "erro". Será que foi mesmo? Não acho. Basta perder o "mal costume" e programar de acordo com a tecnologia escolhida. Pense bem, qual a utilidade de um *browse* exibindo um milhão de registros? O usuário levaria horas ou dias para visualizar todos eles. Não seria muito mais inteligente trazer os registros já "peneirados" por algum critério?

Conclusão

Escolher um banco de dados não é tarefa fácil, nem rápida. Teste todas as opções, usando uma massa de dados reais e num ambiente mais próximo possível do qual será utilizado em produção. Informe-se! Muito se pode aprender nos sites e listas de discussão disponíveis na internet. Livros e cursos também são uma ótima opção.

A conversa com outros usuários para troca de experiências pode ajudar, mas cuidado para não se enganar com depoimentos inverídicos, geralmente causados pela falta de real conhecimento da pessoa que está sendo consultada. Pagar por uma consultoria a fim de esclarecer possíveis dúvidas ou indicar possíveis soluções pode lhe poupar muito tempo e dinheiro, impedindo ou alertando sobre a tomada de decisões erradas.

Avalie esse artigo

ÓTIMO BOM RUIM

Autor:

Carlos Henrique Cantu

Mini-curriculo

Bacharel em Ciência da Computação e pós-graduado em análise de Sistemas. Trabalha com desenvolvimento de sistemas há 16 anos. É consultor especializado em bancos de dados Firebird e InterBase 6.0. Mantenedor do site FireBase (www.firebase.com.br), autor do livro Firebird Essencial e presidente do DUG-BR (Delphi Users Group Brasil), tendo ministrado palestras sobre Firebird para milhares de pessoas em diversas cidade do Brasil. Foi palestrante da Conferência Internacional de Firebird 2005, realizada em Praga (República Tcheca). Também é editor da DB FreeMagazine.

Log de alterações no PostgreSQL

por Mauro Henrique Costa Matos

Atualmente a mídia tem divulgado com freqüência notícias sobre fraudes eletrônicas e de golpes na Internet. Essas ações sempre lesam alguém de alguma forma, gerando algum tipo de perda, geralmente financeira, surgindo a necessidade de desenvolvimento de mecanismos cada vez mais sofisticados pelas organizações, na tentativa de aumentar a segurança das informações.

Em uma investigação, os registros em meios eletrônicos são fundamentais, sendo geralmente armazenados em um SGBD (Sistema de Gerenciamento de Banco de Dados). Nesse artigo, demonstrarei uma solução viável para registrar as modificações dos dados de forma automática em bancos de dados relacionais.

Algumas das vantagens da solução proposta são:

- Permite recuperação imediata dos dados através do uso de instruções SQL;
- Possibilita a criação de aplicativos para verificar a integridade dos dados;
- As aplicações poderão fazer uso das tabelas dos históricos para recuperar os dados alterados por engano, além de possibilitar obter a autoria do fato, responder quando e de onde a operação foi realizada;
- Permite criar uma infra-estrutura rapidamente, sem exigir grandes esforços.

A meta é fazer com que próprio banco de dados se encarregue dos procedimentos necessários para o registro das modificações realizadas nos registros da tabela que se deseja investigar. Esses dados são transferidos antes que ocorra qualquer modificação para uma outra tabela, que manterá o histórico de todas as alterações ou exclusões. Ou seja, o processo todo se resume em criar mecanismos para manter o histórico das modificações realizadas nos dados.

Para tanto, definiremos um modelo para as tabelas de histórico dos registros, que está logicamente organizada em duas partes:

A primeira parte irá conter as informações de controle, enquanto que a segunda é uma cópia idêntica da estrutura da tabela que sofrerá as alterações dos dados.

O modelo para as informações de controle contém as seguintes informações:

Data e hora da atualização;
Endereço IP do cliente;
Identificador do usuário;
Operação realizada;
Hash do novo registro[1] na tabela auditada;
Hash do registro[1] antigo;
Hash do registro[1] corrente (das informações de controle).

[1] Registro é cada linha de uma tabela em um banco de dados.

Para demonstrar a viabilidade dessa tecnologia, vou utilizar o banco de dados PostgreSQL na sua versão 8.x, todavia essa solução poderá ser implementada no Oracle, DB2, SQL Server ou em qualquer outro banco de dados relacional que aceite procedimentos armazenados, sendo necessário apenas algumas adaptações nos scripts, adequando-os a linguagem do SGBD utilizado. O sistema operacional utilizado foi o Linux Fedora Core 3 com o kernel 2.6.14-1.1653_FC4.

Os scripts necessários para a criação das estrutura e procedures podem ser baixados aqui, e podem ser vistos no final do artigo (**listagem 1 e 2**).

O primeiro passo é descompactar o arquivo historico5.zip. Use o seu descompactador preferido. Recomendo que seja criado um diretório específico para essa operação.

No Linux, abra um terminal e digite na linha de comando:

```
1. $ mkdir hist
2. $ cd hist
```

Descompacte o arquivo no diretório hist.

```
1. $ unzip historico5.zip
2. Archive:  historico5.zip
3.   inflating: historico.sql
4.   inflating: leiame.txt
5.   inflating: plpgsql.sql
6. $
```

Para obter maiores detalhes sobre a instalação, recomendo a leitura do arquivo leiame.txt.

Rode agora o terminal interativo do PostgreSQL(PSQL).

```
$ psql <nome_da_base> [-h <host>] [-U
<usuário_do_banco>]
```

Ou simplesmente chame a aplicação seguida do nome do banco. É importante que o "usuário do banco" tenha acesso privilegiado ao banco de dados onde se deseja criar os históricos das tabelas.

```
$ psql <nome do banco> -h localhost
```

Para maiores informações digite:

```
$ psql --help
```

Os comandos a seguir serão digitados na linha de comando do PSQL.

Após entrar no terminal interativo do PostgreSQL, se ainda não tiver sido definido a linguagem plpgsql, digite o comando:

```
=# \i plpgsql.sql
```

Com o objetivo de facilitar a administração, vamos criar um esquema no banco de dados:

```
=# CREATE SCHEMA historico;
```

Em seguida digite os comandos:

```
1. =# SHOW search_path;
2. =# SET search_path TO historico,
   public;
3. =# \i historico.sql
```

Com isso, os scripts que vão criar as rotinas para geração dos históricos já estão no seu banco de dados.

Para exemplificar o uso, vamos criar no seu banco de dados uma tabela chamada **produtos**, com as seguintes definições:

```
1. =# CREATE TABLE public.produtos (
2. (# id serial,
3. (# nome text,
4. (# preco numeric
5. (# )
6. -# WITHOUT OIDS;
```

Ainda no psql, inclua algumas linhas na tabela produtos:

```
1. =# INSERT INTO public.produtos (nome,
   preco) values ('José', 3000.00);
2. =# INSERT INTO public.produtos (nome,
   preco) values ('Beatriz', 2900.00);
3. =# INSERT INTO public.produtos (nome,
   preco) values ('Mauro', 2900.00);
4. =# INSERT INTO public.produtos (nome,
   preco) values ('Paulo', 29000.00);
```

Para criar a estrutura necessária para o registro das modificações em uma tabela, usamos a seguinte sintaxe:

```
SELECT create_trigger(<nome_tab_
auditada>, <nome_tab_historico>);
```

Sendo que o <nome_tab_auditada> (nome da tabela auditada) e o <nome_tab_historico> (nome da tabela de históricos) podem vir precedidas por um qualificador, ou seja, pode ser especificado um esquema.

A instrução SQL abaixo criará além do gatilho, a tabela e o procedimento armazenado responsável pelo registro das modificações realizadas na tabela produtos:

```
=# SELECT create_trigger('public.
produtos','historico.produtos_h');
```

Se tudo correr como previsto, será mostrada uma listagem com várias instruções SQL. A partir desse momento, todas as modificações feitas nos dados dos registros da tabela **produtos**, serão registradas na tabela **protutos_h**.

Para testar, digite o seguinte comando:

```
=# UPDATE produtos set preco=3000 WHERE
preco = 2900;
```

Use o comando a seguir para verificar:

```
=# select * from historico.produtos_h;
```

Para sair do psql:

```
=# \q
```

Considerações finais

Para que se tenha acesso ao esquema "*historico*" nas próximas vezes que o banco for iniciado, edite o arquivo **postgresql.conf** e altere o conteúdo de **search_path** para conter também o esquema desejado.

Lembre-se de que os usuários "comuns" devem ter privilégio de acesso apenas para inclusão nas tabelas de históricos. Ninguém deverá ter acesso para atualização. Apenas as pessoas responsáveis pela manutenção, ou seja, pela remoção dos registros antigos (após um backup), deve ter acesso a remoção de registros.

Excepcionalmente, pode-se permitir consultas aos dados dos históricos para os usuários que possam utilizá-los como base para recuperação de registros das tabelas auditadas.

Conclusão

Apresentamos nesse artigo uma boa solução de mecanismo para registrar modificações nos dados das tabelas de um banco de dados, podendo servir como referência para novas implementações em diferentes SGBDs.

Avalie esse artigo

ÓTIMO	BOM	RUIM

Autor:

Mauro Henrique Costa Matos

Mini-curriculo

Mauro Henrique Costa Matos é formado em Tecnologia em Processamento de Dados (UNEB-DF) e pós-graduado em Redes de Computadores (Católica-DF). Já trabalhou como instrutor, suporte técnico, programador e analista de sistemas. Atualmente trabalha com Linux e ferramentas "open sources". Pode ser contatado através do email mauromatos@ig.com.br.

```
--
--  OBJETIVO:           Cria a infra-estrutura necessárias para geração de
--                      histórico das modificações efetuadas nas tabelas da
--                      base de dados.
--  AUTOR:              MAURO H. C. MATOS
--  DATA DE CRIAÇÃO:    16/09/2002
--  ÚLTIMA ATUALIZAÇÃO: 08/02/2006
--


--
--  PRÉ-REQUISITO:
--  --------------
--  Caso a linguagem plpgsql ainda não tenha sido definida em seu banco de
--  dados, o script plpgsql.sql deve ser rodado antes que se rode este script.
--  O terminal interativo do PostgreSQL (psql) poderá ser utilizado para essa
--  finalidade, lembre-se que o usuário deverá possuir privilégio de
--  "administrador" do banco de dados.
--  A função create_trigger(), assim como todas as demais funções criadas
--  pela mesma, são escrita nessa "linguagem procedural".
--  A infra-estrutura é criada no esquema (espaço de tabela) corrente.
--


--
-- CRIA a FUNÇÃO get_fields
--
CREATE OR REPLACE FUNCTION get_fields (text) RETURNS text[] AS $$
DECLARE
--
--  NOME:               get_fields
--  AUTOR:              MAURO H. C. MATOS
--  DATA DE CRIAÇÃO:    07/10/2005
--  ÚLTIMA ATUALIZAÇÃO: 08/02/2006
--  OBJETIVO:           RECUPERAR OS NOMES DOS CAMPOS DE UMA TABELA.
--  PARÂMETROS:         1=NOME DE TABELA COM OU SEM QUALIFICADOR
--  RETORNA:            ARRAY COM OS NOMES DOS CAMPOS OU NULO.
--  ------------------------------------------------------------------------------
--
    pNomeTab        alias for $1; -- nome da tabela com/sem qualificador
    aSQL            text;         -- instrução SQL
    aNomeEsqTab     text;         -- nome da esquema da tabela
    aNomeTab        text;         -- nome da tabela
    iQtd            integer;      -- usado em operações
    rIdRec          RECORD;       -- identificador de registro
    aVetor          text[];       -- vetor de retorno
BEGIN
    --
    -- VERIFICA OS PARÂMETROS
    --
    IF pNomeTab = '' THEN
        RAISE EXCEPTION '02001:Não pode haver parâmetro vazio.';
    END IF;
    --
    -- VERIFICA A FORMAÇÃO DOS NOME DAS TABELAS E RECUPERA OS CAMPOS
    --
    iQtd := strpos( pNomeTab, '.');
    IF iQtd > 0 THEN
        -- existe o nome do qualificador da tabela
        aNomeEsqTab := substr(pNomeTab, 1, iQtd - 1); -- nome do qualificador
        aNomeTab := substr(pNomeTab, iQtd + 1);       -- nome da tabela
        -- instrução SQL para recuperar os nomes dos campos da tabela
        aSQL := 'SELECT column_name' ||
                ' FROM information_schema.columns' ||
              ' WHERE table_name = \'' || aNomeTab || '\'' ||
                ' AND table_schema = \'' || aNomeEsqTab  || '\'' ||
              ' ORDER BY ordinal_position';
    ELSE
        aNomeEsqTab := NULL;
        -- verifica se existe mais de uma tabela com o mesmo nome no banco
        SELECT count(*)
        FROM pg_tables INTO iQtd
        WHERE tablename = pNomeTab;
        IF iQtd <> 1 THEN   -- existe mais de uma tabela com o mesmo nome
            RAISE EXCEPTION
            '02021:Especifique o qualificador da tabela %', pNomeTab;
        END IF;
        -- instrução SQL para recuperar os nomes dos campos da tabela
        aSQL := 'SELECT column_name' ||
                ' FROM information_schema.columns' ||
                ' WHERE table_name = \'' || pNomeTab || '\'' ||
                ' ORDER BY ordinal_position';
    END IF;
    --
    -- MONTA O ARRAY
    --
    iQtd = 0; -- inicializa contador
    FOR rIdRec IN EXECUTE aSQL LOOP
        iQtd := iQtd + 1;
        aVetor[iQtd] := rIdRec.column_name::text;
    END LOOP;
    IF iQtd > 0 THEN
        RETURN aVetor; -- retorna o vetor
    ELSE
        RETURN NULL;   -- retorna nulo
    END IF;
END;
$$ LANGUAGE plpgsql STABLE STRICT;

COMMENT ON FUNCTION get_fields (text)
    IS 'Retorna os nomes dos campos de uma tabela';

--
-- REMOVE A TABELA history_template
--
DROP TABLE history_template;

--
-- CRIA A TABELA history_template
--
CREATE TABLE history_template (
    _date_          timestamp
                    with time zone, -- data e hora da atualização
    _local_         inet,           -- IP do local da alteração
    _user_          name,           -- identificador do usuário
    _op_            char(1),        -- operação (U=Update ou D=Delete)
    _ctrl_newrec_   text,           -- hash do novo registro na tabela auditada
    _ctrl_oldrec_   text,           -- hash do registro antigo
    _ctrl_audit_    text            -- hash do registro corrente
);
COMMENT ON TABLE history_template
    IS 'Modelo para montar as tabelas de histórico';

--
-- CRIA a FUNÇÃO create_trigger
--
CREATE OR REPLACE FUNCTION create_trigger (text, text) RETURNS void AS $$
DECLARE
--
--  NOME:               create_trigger
--  AUTOR:              MAURO H. C. MATOS
--  DATA DE CRIAÇÃO:    05/02/2004
--  ÚLTIMA ATUALIZAÇÃO: 02/11/2005
--  OBJETIVO:           CRIA A FUNÇÃO E O GATILHO PARA GERAR HISTÓRICO DE UMA
--                      TABELA.
--  PARÂMETROS:         1=NOME DA TABELA QUE SERÁ AUDITADA, OPCIONALMENTE PODE
--                        SER ESPECIFICADO O NOME DO ESQUEMA.
--                      2=NOME DA TABELA QUE SERÁ CRIADA PARA CONTER O
```

```
--                      HISTÓRICO, OPCIONALMENTE PODE SER ESPECIFICADO O
--                      NOME DO ESQUEMA.
--
    pTabAudit       alias for $1; -- nome da tabela auditada
    pTabHist        alias for $2; -- nome da tabela de histório a ser criada
    aNomeEsqAudit   text;         -- nome da esquema da tabela auditada
    aNomeEsqHist    text;         -- nome do esquema da tabela a ser criada
    aNomeTabAudit   text;         -- nome da tabela sem qualificador
    aNomeTabHist    text;         -- nome da tabela sem qualificador
    aNomeFunc       text;         -- nome da função
    aNomeGatilho    text;         -- nome do gatilho
    aSQL            text;         -- instrução SQL
    aSQL2           text;         -- complemento de instrução SQL
    iQtd            integer;      -- valor de retorno
    aCampos         text[];       -- array com o nome dos campos da tabela
    aDim            text;         -- dimensões do array com nomes dos campos
    iMax            integer;      -- indice máximo do array
    aCampo          text;         -- nome de um campo
BEGIN
    --
    -- VERIFICA OS PARÂMETROS
    --
    IF pTabAudit = '' OR pTabHist = '' THEN
        RAISE EXCEPTION '02001:Não pode haver parâmetro vazio.';
    END IF;

    --
    -- VERIFICA SE PARÂMETROS SÃO DIFERNTES
    --
    IF pTabAudit = pTabHist THEN
        RAISE EXCEPTION '02002:Deverá ser especificados parâmetros diferentes.';
    END IF;

    --
    -- VERIFICA A FORMAÇÃO DOS NOME DAS TABELAS
    --
    iQtd := strpos( pTabAudit, '.');
    IF iQtd > 0 THEN
        -- existe o identificador de esquema da tabelas
        aNomeEsqAudit := substr(pTabAudit, 1, iQtd - 1);
        aNomeTabAudit := substr(pTabAudit, iQtd + 1);
    ELSE
        aNomeEsqAudit := NULL;
        aNomeTabAudit := pTabAudit;
    END IF;
    iQtd := strpos( pTabHist, '.');
    IF iQtd > 0 THEN
        -- existe o identificador de esquema da tabelas
        aNomeEsqHist := substr(pTabHist, 1, iQtd - 1);
        aNomeTabHist := substr(pTabHist, iQtd + 1);
        -- monta o nome da função
        aNomeFunc := aNomeEsqHist || '.' || aNomeTabAudit || '_audit()';
    ELSE
        aNomeEsqHist := NULL;
        aNomeTabHist := pTabHist;
        -- monta o nome da função
        aNomeFunc := pTabAudit || '_audit()';
    END IF;

    --
    -- VERIFICA A EXISTÊNCIA DA DEFINIÇÃO DA LINGUAGEM
    --
    SELECT count(*) FROM pg_catalog.pg_language  INTO iQtd
    WHERE lanname = 'plpgsql';
    IF iQtd = 0 THEN
         RAISE EXCEPTION '02010:A linguagem plpgsql ainda não foi definida.';
    END IF;

    --
    -- VERIFICA A EXISTÊNCIA DA TABELA history_template
    --
    SELECT count(*) FROM pg_catalog.pg_tables  INTO iQtd
    WHERE tablename = 'history_template';
    IF iQtd = 0 THEN
         RAISE EXCEPTION '02003:A tabela history_template não exite.';
    END IF;

    --
    -- VERIFICA A EXISTÊNCIA DA TABELA A SER AUDITADA
    --
    IF aNomeEsqAudit IS NULL THEN
        SELECT count(*) FROM pg_tables INTO iQtd
        WHERE tablename = aNomeTabAudit;
    ELSE
        SELECT count(*) FROM pg_tables INTO iQtd
        WHERE schemaname = aNomeEsqAudit
        AND tablename = aNomeTabAudit;
    END IF;
    IF iQtd = 0 THEN
        RAISE EXCEPTION '02004:A tabela % não exite.', aNomeTabAudit;
    END IF;

    --
    -- CRIA A NOVA TABELA PARA ARMAZENAR O HISTÓRICO DA TABELA AUDITADA
    --
    IF aNomeEsqHist IS NULL THEN
        SELECT count(*) FROM pg_tables INTO iQtd
        WHERE tablename = aNomeTabHist;
    ELSE
        SELECT count(*) FROM pg_tables INTO iQtd
        WHERE schemaname = aNomeEsqHist
        AND tablename = aNomeTabHist;
    END IF;
    IF iQtd = 0 THEN
        -- a tabela ainda não existe
        aSQL := 'CREATE TABLE ' || pTabHist || ' WITHOUT OIDS AS ' ||
            'SELECT * FROM history_template CROSS JOIN ' || pTabAudit ||';';
        RAISE NOTICE '%', aSQL; -- mostra o comando create
        EXECUTE aSQL;
        IF NOT FOUND THEN
            RAISE EXCEPTION '02005:Não foi possível criar a tabela %.', pTabHist;
        END IF;
        -- comantário para a tabela criada
        aSQL := 'COMMENT ON TABLE ' || pTabHist ||
            ' IS \'Histórico da tablela ' || pTabAudit || '\';';
        RAISE NOTICE '%', aSQL; -- mostra o comando comment
        EXECUTE aSQL;
        -- permissões para os usuários incluirem dados na tabela criada
        aSQL := 'GRANT INSERT ON TABLE ' || pTabHist || ' TO PUBLIC;';
        RAISE NOTICE '%', aSQL; -- mostra o comando grant
        EXECUTE aSQL;
        IF NOT FOUND THEN
            RAISE EXCEPTION
            '02006:Não foi possível conceder privilégios para %.', pTabHist;
        END IF;
    ELSE
        aSQL := '\n\n===> ATENÇÃO: A tabela ' || pTabHist || ' já exite, ' ||
        'irá ocorrer erro\n ao tentar atualizar ou remover dados na tabela ' ||
        pTabAudit || ',\n se tiver havido alteração na estrutura da mesma.\n';
        RAISE NOTICE '%', aSQL;
    END IF;

    --
    -- MONTA A FUNÇÃO CHAMADA PELO GATILHO
    --
    -- início da primeira parte da função
```

```
273.    aSQL :=
274.    '\nCREATE OR REPLACE FUNCTION ' || aNomeFunc ||
275.    ' RETURNS trigger AS $BODY$ ' ||
276.    '\nDECLARE' ||
277.    '\n    tNow        timestamp with time zone;  -- data e hora' ||
278.    '\n    iLocal      inet;     -- identificador da máquina cliente' ||
279.    '\n    aUsr        name;     -- identificador do usuário' ||
280.    '\n    cOp         char;     -- identificador da operação' ||
281.    '\n    aCtrlNewRec text;     -- hash do novo registro da tab. auditada' ||
282.    '\n    aCtrlOldRec text;     -- hash do antigo registro da tab. auditada' ||
283.    '\n    aCtrlAudit  text;     -- hash para o registro da tab do histórico' ||
284.    '\n    aData       text[];   -- dado de um campo' ||
285.    '\n    aDataFields text;     -- dados dos campos' ||
286.    '\n    iQtd        integer; -- quantidade' ||
287.    '\nBEGIN' ||
288.    '\n    tNow := now();' ||
289.    '\n    iLocal := inet_client_addr();' ||
290.    '\n    aUsr := current_user;' ||
291.    '\n    cOp := substr(TG_OP, 1, 1); -- inicial da descrição da operação' ||
292.    '\n    aDataFields := \'\'; -- limpa a variável' ||
293.    '\n    iQtd := 0;';
294.    -- fim da primeira parte da função
295.
296.    -- início da segunda parte da função
297.    aCampos := get_fields(pTabAudit); -- nome dos campos
298.    aDim := array_dims(aCampos);      -- dimensões do array
299.    iMax := replace(split_part(aDim,':',2),']','')::int;
300.    -- monta complemento da instrução SQL
301.    aSQL2 := '';
302.    FOR iX IN 1 .. iMax LOOP
303.        aCampo :=  'OLD.' || aCampos[iX];
304.        aSQL2 := aSQL2 ||
305.        '\n    IF ' || aCampo || ' IS NOT NULL THEN' ||
306.        '\n        iQtd := iQtd + 1;' ||
307.        '\n        aData[iQtd] := ' || aCampo || ';' ||
308.        '\n    END IF;';
309.    END LOOP; -- fim do laço FOR
310.    -- fim da segunda parte da função
311.
312.    aSQL := aSQL || aSQL2; -- monta a segunda parte na instrução SQL
313.
314.    -- início da terceira parte da função
315.    aSQL2 :=
316.    '\n    -- calcula o hash do registro antigo' ||
317.    '\n    FOR iX IN 1..iQtd LOOP' ||
318.    '\n        aDataFields := aDataFields || aData[iX];' ||
319.    '\n    END LOOP;' ||
320.    '\n    aCtrlOldRec := md5(aDataFields);' ||
321.    '\n    IF (cOp = \'U\') THEN         -- UPDATE' ||
322.    '\n       iQtd := 0;';
323.    -- fim da terceira parte da função
324.
325.    aSQL := aSQL || aSQL2; -- monta a terceira parte na instrução SQL
326.
327.    -- início da quarta parte da função
328.    aSQL2 := '';
329.    FOR iX IN 1 .. iMax LOOP
330.        aCampo :=  'NEW.' || aCampos[iX];
331.        aSQL2 := aSQL2 ||
332.        '\n        IF ' || aCampo || ' IS NOT NULL THEN' ||
333.        '\n            iQtd := iQtd + 1;' ||
334.        '\n            aData[iQtd] := ' || aCampo || ';' ||
335.        '\n        END IF;';
336.    END LOOP; -- fim do laço FOR
337.    -- fim da terceira parte da função
338.
339.    aSQL := aSQL || aSQL2; -- monta a quarta parte na instrução SQL
340.
```

```
341.    -- início da quinta parte da função
342.    aSQL := aSQL ||
343.    '\n        aDataFields := \'\'; -- limpa a variável' ||
344.    '\n        FOR iX IN 1..iQtd LOOP' ||
345.    '\n            aDataFields := aDataFields || aData[iX];' ||
346.    '\n        END LOOP;' ||
347.    '\n        aCtrlNewRec := md5(aDataFields);' ||
348.    '\n        aDataFields := tNow::text || iLocal::text || aUsr || cOp' ||
349.    '\n                       || aCtrlNewRec || aCtrlOldRec;' ||
350.    '\n        aCtrlAudit := md5(aDataFields);' ||
351.    '\n        INSERT INTO ' || pTabHist ||
352.    '\n            SELECT tNow,' ||
353.    '\n                   iLocal,' ||
354.    '\n                   aUsr,' ||
355.    '\n                   cOp,' ||
356.    '\n                   aCtrlNewRec,' ||
357.    '\n                   aCtrlOldRec,' ||
358.    '\n                   aCtrlAudit,' ||
359.    '\n                   OLD.*;' ||
360.    '\n        RETURN NEW;' ||
361.    '\n    ELSIF (cOp = \'D\') THEN      -- DELETE' ||
362.    '\n        aCtrlNewRec := \'\';      -- sem conteúdo' ||
363.    '\n        aDataFields := tNow::text || iLocal::text || aUsr || cOp' ||
364.    '\n                       || aCtrlNewRec || aCtrlOldRec;' ||
365.    '\n        aCtrlAudit := md5(aDataFields);' ||
366.    '\n        INSERT INTO ' || pTabHist ||
367.    '\n            SELECT tNow,' ||
368.    '\n                   iLocal,' ||
369.    '\n                   aUsr,' ||
370.    '\n                   cOp,' ||
371.    '\n                   aCtrlNewRec,' ||
372.    '\n                   aCtrlOldRec,' ||
373.    '\n                   aCtrlAudit,' ||
374.    '\n                   OLD.*;' ||
375.    '\n        RETURN OLD;' ||
376.    '\n    ELSE' ||
377.    '\n        RETURN NULL;' ||
378.    '\n    END IF;' ||
379.    '\nEND;' ||
380.    '\n$BODY$ LANGUAGE plpgsql STRICT;';
381.    -- fim da função
382.
383.    --
384.    -- CRIA A FUNÇÃO NA BASE DE DADOS
385.    --
386.    RAISE NOTICE '%', aSQL; -- mostra a função criada
387.    EXECUTE aSQL;
388.    IF NOT FOUND THEN
389.        RAISE EXCEPTION '02007:Não foi possível criar a função %.', aNomeFunc;
390.    END IF;
391.    -- comantário para a função criada
392.    aSQL := 'COMMENT ON FUNCTION ' || aNomeFunc ||
393.        ' IS \'Registra modificações na tablela ' || pTabAudit || '\';';
394.    RAISE NOTICE '%', aSQL; -- mostra o comando comment
395.    EXECUTE aSQL;
396.
397.    --
398.    -- PERMISSÕES PARA OS USUÁRIOS EXECUTAREM A FUNÇÃO
399.    --
400.    aSQL := 'GRANT EXECUTE ON FUNCTION ' || aNomeFunc || ' TO PUBLIC;';
401.    RAISE NOTICE '%', aSQL; -- mostra o comando grant
402.    EXECUTE aSQL;
403.    IF NOT FOUND THEN
404.        RAISE EXCEPTION '02008:Não foi possível conceder privilégios para %.',
405.            aNomeFunc;
406.    END IF;
407.
408.    --
```

```
409.    -- MONTA O NOME DO GATILHO
410.    --
411.    aNomeGatilho := aNomeTabHist || '_audit';
412.
413.    --
414.    -- REMOVE O GATILHO SE EXISTIR
415.    --
416.    SELECT count(*) FROM pg_trigger INTO iQtd
417.    WHERE tgname =  aNomeGatilho;
418.    IF iQtd > 0 THEN
419.        -- exite trigger definida
420.        aSQL := 'DROP TRIGGER ' || aNomeGatilho || ' ON '
    || pTabAudit ||
421.            ' CASCADE;';
422.        RAISE NOTICE '%', aSQL; -- mostra o comando drop
423.        EXECUTE aSQL;
424.    END IF;
425.
426.    --
427.    -- CRIA O GATILHO QUE DISPARA A FUNÇÃO CRIADA
428.    --
429.    aSQL := 'CREATE TRIGGER ' || aNomeGatilho ||
430.        ' BEFORE UPDATE OR DELETE ON '  || pTabAudit ||
431.        ' FOR EACH ROW EXECUTE PROCEDURE ' || aNomeFunc;
432.    RAISE NOTICE '%', aSQL; -- mostra o comando create
    trigger
433.    EXECUTE aSQL;
434.    IF NOT FOUND THEN
435.        RAISE EXCEPTION '02009:Não foi possível criar o
    gatilho %.',aNomeGatilho;
436.    END IF;
437.    RAISE NOTICE 'create_trigger() - Versão 5
    (NOV/2005)'; -- mensagem final
438.    RETURN;
439.END;
440.$$ LANGUAGE plpgsql STRICT;
441.
442.COMMENT ON FUNCTION create_trigger (text, text)
443.    IS 'Cria a infra-estrutura necessária para gerar
    histórico de uma tabela';
```

Listagem 1. Conteúdo do script historico.sql

```
1.  /*
2.      OBJETIVO:     INSTALAR A LINGUAGEM PL/pgSQL NO BANCO
    DE DADOS.
3.      AUTOR:        MAURO H. C. MATOS
4.      CRIAÇÃO:      13/09/2002
5.      ATUALIZAÇÃO: 21/04/2005
6.  */
7.
8.  -- REGISTRA A FUNÇÃO TRATADORA DE CHAMADAS
9.  CREATE OR REPLACE FUNCTION plpgsql_call_handler() RETURNS
    language_handler
10.     AS '$libdir/plpgsql'
11.     LANGUAGE C;
12.
13. COMMENT ON FUNCTION plpgsql_call_handler()
14.     IS 'Registra o tratador de chamadas associado
    linguagem plpgsql';
15.
16. -- CRIA A LINGUAGEM PL/pgSQL
17. CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
18.     HANDLER plpgsql_call_handler;
```

Listagem 2. Conteúdo do script plpgsql.sql

Famosos e gratuitos

por Luiz Paulo de O. Santos

Já faz algum tempo que utilizo bancos de dados Open-Source e gratuitos, mas os últimos acontecimentos têm me chamado a atenção:

Atualmente é possível baixar versões funcionais (e não de avaliação ou teste) de praticamente todos os grandes bancos de dados, logicamente com restrições, mas funcionais, que teoricamente não expiram e podem ser usados em aplicações, em ambiente de produção e em alguns casos até mesmo podem ser distribuídos em conjunto com uma aplicação que o desenvolvedor produza, e gratuitamente.

Logo, algumas questões surgem nos meus pensamentos:

- Será que se os bancos de dados Open-Sources não existissem esse fenômeno estaria acontecendo?
- Será que os grandes bancos de dados estão incomodados pelos bancos Open-Sources?
- Será que as companhias continuarão à distribuir gratuitamente versões de seus bancos de dados se o uso dos bancos Open-Source diminuir significativamente?
- Ou qual seria a(s) outra(s) razão(ões) pela qual as grandes companhias estão distribuindo gratuitamente seus bancos de dados? A concorrência entre elas? Ou a concorrência com os Open-Sources?

Certamente o fato de uma companhia qualquer ter lançado uma versão gratuita de seu SGBD deve ter provocado nos concorrentes o sentimento da necessidade de fazer algo semelhante, causando um efeito "bola de neve". Mas impulsionada pelo o que, ou porquem? Por que será que a primeira companhia lançou sua versão gratuita.

Sou de um tempo (não tão distante, e não de uma galáxia longínqua) que a única forma de aprender a usar bancos de dados cliente/servidor era adquirindo a ferramenta no mercado através de licenças de desenvolvedor, ou seja, tínhamos que pagar para aprender e desenvolver. Meu cliente também pagava, obviamente um valor muito maior, para poder utilizar o banco em produção.

Em alguns casos, tínhamos uma situação ainda pior: nós, desenvolvedores, freqüentávamos feiras e convenções onde nos era ofertado CDs com versões *trial*, que podíamos usar por um período x (quase sempre inferior a três meses). Quando o prazo expirava, tínhamos que formatar o computador e reinstalar a versão *trial* novamente. Quem nunca fez isso? O resultado após 2 ou três intervenções no micro era trocar o banco de dados.

Particularmente, "peguei ódio" de dois bancos de dados que são considerados fortes no mercado devido à essa situação. Não as usarei nem que me liberem licenças completas e gratuitas ad-eternum. E foi exatamente por isso que passei a utilizar bancos Open-Source. Se pararem de desenvolver não têm problema, tenho o código fonte e posso continuar a usar o banco.

Mas a situação está mudando. As grandes companhias, agora, nos querem ao seu lado. Nos enviam convites para eventos em papel e envelopes caros, nos oferecem coffee-breaks com produtos caros e importados, nos presenteiam com pastas e CDs com versões reduzidas e gratuitas de seus bancos de dados. Nos tratam com o maior carinho e como se fossemos importantes. *E realmente somos!* Pois somos nós quem administra e implementa sistemas usando seus bancos de dados.

E agora eu me pergunto: Qual será a razão disso tudo estar acontecendo? Novamente: Será que os bancos Open-Source têm algo a ver com isso?

Indiferente do que eu penso à respeito desse ou daquele banco de dados, como são gratuitos, cabe mostrar à vocês um pouco sobre cada um dos três produtos que escolhi para falar nesse artigo.

1. IBM DB2® Universal Database™ Express Edition for Linux and Windows.

IBM DB2® Universal Database™ Express Edition V8.2 combina o poder, as funções e confiabilidade de um servidor de banco de dados baseado em padrões abertos com simplicidade, em um pacote de instalação e desenvolvimento com um custo mínimo de investimento.

"IBM introduz o DB2 Express-C, uma versão do DB2 Universal Database Express Edition (DB2 Express) para a comunidade. DB2 Express-C é um notável servidor de dados para uso no desenvolvimento e distribuição com aplicativos. Contém o mesmo Data-Core e características do DB2 Universal Database Express Edition, DB2 Express-C oferece uma sólida base para construir e desenvolver todo tipo de aplicações incluindo: C/C++, Java, .NET, PHP, e mais."

Limitações da versão Express:

- Pode ser instalado em máquinas com no máximo 2 processadores;
- Utiliza no máximo 4GB de RAM;
- Não possibilita particionamento do BD;
- Não possui o DB2 Query Patroller, Net Search Extender,

O Link para o artigo e download é:

http://www-306.ibm.com/software/data/db2/udb/db2express/

2. Microsoft SQL 2005 Express Edition

O SQL Server Express é gratuito e fácil de usar - uma versão do SQL Server 2005 projetada para construir aplicações simples. Os desenvolvedores podem projetar *schemas*, acrescentar dados

e executar *queries* nos bancos de dados locais, tudo dentro do ambiente do *Visual Studio 2005*. Se os desenvolvedores necessitarem de recursos mais sofisticados, podem facilmente migrar para as versões comerciais do SQL Server.

Se for distribuir o MSDE para seus clientes em aplicações que tenha desenvolvido, é necessário registrá-lo (informar quem estará utilizando) para poder utilizar sem ter que pegar por isso.

Algumas limitações dessa versão são:

- Suporte à apenas 1 CPU, mesmo em sistemas que tenham mais de um processador instalado;
- Limite de 1GB de RAM para ser usado para queries e páginas de dados;
- Bancos de dados limitados em 4GB de tamanho;
- Não há serviços de análise ou de relatórios, bem como Data mining, Data Transformation Services (DTS);
- Não há possibilidade de Clustering ou mirroring, pesquisa e indexação de "full-text", SQLMail, indexed views, partitioned views e SQL Agent.

Maiores informações e downloads no endereço abaixo:

http://www.microsoft.com/sql/editions/express/default.mspx

3. Oracle

A Oracle possui dois pacotes gratuítos:

3.1. Oracle Database 10g Express Edition

A versão Express do Oracle agora é gratuíta. O Oracle Database 10g Express Edition é uma versão focada nos desenvolvedores de aplicações, administradores de banco de dados, fornecedores de software e estudantes da área.

O Oracle Database XE (**Ex**press **E**dition) é completamente compatível com a família de bancos de dados da Oracle, que inclui o *Oracle Standard Edition One*, *Oracle Standard Edition* e *Oracle Enterprise Edition*. Os usuários podem começar com um projeto pequeno e, quando necessário, podem fazer o upgrade do *Oracle Database XE* em suas aplicações para outras versões do banco de dados *Oracle 10g*, sem a necessidade de extremas mudanças, trocando apenas o gerenciador.

O Oracle Database XE possui interfaces SQL e PL/SQL, também disponíveis em outras versões do Oracle. Ele também dispõe de suporte para o desenvolvimento de aplicações Java, .NET, PHP em Windows ou Linux. Traz, ainda, o recurso Oracle HTML DB, para rápido desenvolvimento e implementação de aplicativos para web.

Algumas limitação da versão XE:

- Armazenará no **máximo 4GB** de dados;
- Utiliza até **1GB de memória RAM** no servidor;
- Usa no máximo um processador, indiferente da quantidade de processadores instalados.

Downloads no link abaixo:

http://www.oracle.com/technology/products/database/xe/index.html

3.2. Oracle Database Personal Edition

Oracle Database Personal Edition é uma versão do banco de dados Oracle para uso individual, que implementa todas as características encontradas na família Oracle de banco de dados. É fácil de usar, com replicação avançada e características de distribuição que você poderá facilmente trabalhar em um ambiente Oracle Enterprise.

Projetado para desenvolvimento (single user), Personal Edition têm todas características e recursos do Oracle Enterprise Edition, e isso a torna ideal para desenvolvedores. Maiores informações e o download pode ser obtido na URL:

http://www.oracle.com/database/Personal_Edition.html

Conclusão

Embora haja versões gratuítas de bancos de dados famosos, ainda prefiro utilizar bancos OpenSource em minhas aplicações, pois não tenho garantias de até quando essas versões serão realmente gratuítas, ou se há compromisso em mante-las atualizadas com novos releases.

Não sou contra a utilização desses SGBDs, afinal são comerciais e precisam vender, mas o que me preocupa é ficar desamparado no futuro, e ter que passar a pagar para usar um banco de dados ou ter que adequar todas as minhas aplicações para outro SGBD.

Autor:

Luiz Paulo de Oliveira Santos

Mini-curriculo

Luiz Paulo de Oliveira Santos é formado em Tecnologia de Processamento de Dados, especialista em Análise de Sistemas e Redes de Computadores. É analista de suporte de redes na Universidade Metodista de Piracicaba e diretor da JobVox Sistemas Informatizados. Contato: lpaulo@dbfreemagazine.com.br

Primeiros passos com a ZeosLib

Autor: Mário Lúcio Gomes de Faria

INTRODUÇÃO

A biblioteca ZeosLib é um conjunto de componentes que implementa acessos a diversos bancos de dados, tais como: *MySQL, PostgreSQL, Interbase, Firebird, MS SQL, Sybase, Oracle* e *DB/2*, SQLite, podendo ser utilizada com *Delphi, Kylix, C++ Builder* e com o *Lazarus* (ver quadro em anexo sobre o Lazarus).

A ZeosLib tem licença GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL) e o site oficial do projeto é http://sourceforge.net/projects/zeoslib. Atualmente é administrado por *Frank Linde* e *Michael Seeger*, sendo que a última versão para download é a ZeosDBO 6.5.1 – Alpha, liberada em 13/10/2005

Tecnicamente a ZeosLib ZDBC é a base de todos dos objetos da ZeosLib DBO, sendo que a ZDBC foi implementada nos moldes da JDBC (Java Database Connectivity). Ou seja, aqueles que possuem algum conhecimento da JDBC irão se sentir em casa para utilizar a biblioteca de classes da ZeosLib.

Por sua vez, o ZeosLib DBO manteve certa compatibilidade com os componentes de acesso a banco de dados da Borland (a falecida BDE), facilitando sua utilização conforme será mostrado neste artigo.

INSTALAÇÃO

O processo de instalação dos componentes desta biblioteca é muito simples. Para exemplificar o processo, utilizarei o Delphi 7. Os exemplos utilizam o Firebird pelo simples fato de ser o BD disponível aqui no momento. Nada impede que você use qualquer um dos outros SGBDs suportados.

O primeiro passo é efetuar o download do arquivo **zeosdbo-6.5.1-alpha.zip** no site do projeto, e descompactá-lo em um diretório de sua escolha. Outra opção é fazer o download direto do repositório CVS (Concurrent Version System). Utilizando o repositório CVS para download, a estrutura de diretórios deverá ser algo semelhante a:

```
-- zeos ----------
                |--- database
                |--- documentation
                |--- examples
                |--- lib
                |--- packages
                |--- src
                |--- test
```

Segue uma explicação rápida sobre os diversos subdiretórios:

zeos/database: Este diretório possui diversos arquivos de comandos sql para criar um banco de dados de exemplo. Você poderá escolher qual SGBD pretende utilizar. Além disso, poderá criar também uma massa de dados para testes.

zeos/documentation: Este diretório é muito importante, pois nele encontraremos diversas documentações relacionadas a instalação, correções de erros, dicas, etc.

zeos/examples: Aqui encontraremos diversos arquivos de exemplos de programas escritos em Delphi fazendo uso da ZeosLib. Devo salientar neste momento que, embora simples, estes exemplos podem ser muito úteis no aprendizado. Desta forma, aconselho a todos que procurem compilar e executá-los. Se tiverem interesse em aprender um pouco de programação avançada em Delphi e POO (Programação Orienta a Objeto), vale a pena executar os exemplos em modo de depuração. Os programas fontes desta biblioteca estão muito bem escritos e legíveis.

zeos/lib: Contém as "dlls" para acesso aos bancos de dados MySQL e PostgreSQL

zeos/packages: Aqui encontraremos os diversos arquivos "*.dpk" em subdiretórios que correspondem as diversas versões do compilador Delphi, Cbuilder e FPK com Lazarus.

zeos/src: Contém os programas fontes da biblioteca. Antes de iniciar a instalação, verifiquem o arquivo "Zeos.inc". Nele encontra-se diversas opções de compilação. Para os programadores mais avançados, pode ser bom dar uma olhada.

zeos/test: Contém diversos projetos que são utilizados para testes da biblioteca pelos desenvolvedores.

Feito o download e estando criada uma estrutura de diretórios semelhante a citada anteriormente, veremos resumidamente como é o processo de instalação. Os procedimentos a seguir foram testados no **Delphi 7**, porém, com poucas adaptações servirá também para outras versões.

1. Copiar as dlls do subdiretório "*zeos/lib*" para o diretório de sistema do windows, normalmen-

Lazarus e FreePascal

O "FreePascal" (denominado para os íntimos de FPK) é um compilador Pascal de 32 e 64 bits, profissional, que roda nos sistemas operacionais *Linux, FreeBSD, Mac, DOS, Win32, Netware e MorphOS* e esta disponível para os processadores *Intel x86, Amd64/x86 / 64, PowerPC* e *Sparc*.

A sintaxe do FPK é extremamente compatível com Turbo Pascal 7.0 da Borland Corp. e também mantém muita compatibilidade com o **Object Pascal** (do Delphi), implementando os conceitos de objetos, classes, exceções, interfaces, etc.

O site oficial é www.freepascal.org, sendo a versão 2.0.2 a última estável disponível para download.

Já o Lazarus é um projeto que implementa um conjunto de classes que emulam a IDE e a VCL do Delphi.

O site oficial do Lazarus é www.lazarus.freepascal.org. Ainda não implementa toda a funcionalidade presente no Delphi, mas posso afirmar com toda convicção que logo será mais que somente uma emulação do Delphi.

Tanto o FPK quanto o Lazarus são projetos OPEN SOURCE.

te *windows/system* ou *windows/system32*
2. Adicionar o subdiretório "*zeos/packages/Delphi7*" no *Library Path,* conforme a versão do compilador escolhido.
3. Abrir o arquivo *ZeosDbo.bpg* que deverá ser encontrado no diretório "*zeos/packages/delphi7*" e compilar os componentes nesta ordem:

    1. ZCore.bpl
    2. ZParseSql.bpl
    3. ZPlain.bpl"
    4. Zdbc.bpl

Ou selecionar a opção de compilar todos os componentes.

Se todas as compilações terminaram sem erros, então basta instalar o pacote *Zcomponent.bpl*. Após este passo, todos os componentes ZEOS estarão disponíveis para serem utilizados na barra de componentes do Delphi.

OS COMPONENTES VISUAIS

A versão que foi utilizada neste artigo foi a 6.5.1, contendo os seguintes componentes:

ZConnection

Este componente é utilizado para estabelecer uma conexão com um banco de dados. Ele é obrigatório na aplicação, podendo existir um ou mais em um mesmo formulário. Nele deverá ser parametrizado em qual banco de dados a aplicação irá conectar, qual o tipo de banco de dados, onde se localiza (endereço IP), usuário e senha para conexão com o banco, etc. Uma opção interessante neste componente é a possibilidade de utilizar a conexão somente para leitura (gerando um ganho de "performance" em geral).

Normalmente este componente é colocado somente no formulário principal da aplicação para garantir que somente uma conexão será realizada com o banco de dados.

ZReadOnlyQuery / ZQuery / ZTable

Estes três componentes implementam cada um deles um "Dataset", que poderá ser associado a um "Datasource" presente na aba "Data Access" padrão do Delphi. Os dois primeiros são utilizados principalmente quando o programador deseja ter maior flexibilidade nos comandos de consulta ao banco de dados. O componente "ZTable" implementa um "Dataset" muito semelhante ao componente "TTable" do BDE.

Basicamente, a diferença entre ZReadOnlyQuery e ZQuery é o fato do primeiro ser somente para leitura, ou seja, não possibilita edição de dados.O mais utilizado nas aplicações é o ZQuery.

Após ser definido qual conexão será utilizada com o banco, e o comando SQL, ele passa a se comportar de maneira semelhante ao "TTable" do BDE.

Nos casos em que o comando SQL incluir algum tipo de "join", será necessário utilizar o componente ZUpdateSQL para tornar o dataset editável.

ZUpdateSQL

Este componente deverá ser ligado ao componente ZQuery ou ZTable quando for necessário explicitar os comandos para Apagar, Inserir ou Alterar dados em uma tabela. Sua utilização com o componente ZTable é opcional. Este componente gera eventos antes e depois de cada execução de um comando SQL, oferecendo grande flexibilidade para a aplicação.

ZStoredProc

Usado na execução de *stored procedure*.

ZSQLMetadata

Através dele fazemos acesso ao dicionário de dados do banco de dados de forma independente de qual seja o SGBD utilizado.

ZSQLProcessor

Este componente implementa um processamento em lote, onde diversos comandos podem ser executados seqüencialmente no banco de dados, como um script.

ZSQLMonitor

É utilizado basicamente em fase de depuração da aplicação para "ver" a comunicação entre a aplicação e o SGBD.

O COMPONENTE ZQUERY

Descreverei brevemente como utilizar este componente via programação (não em design-time) para mostrar como ele é flexível e simples, sendo ao mesmo tempo robusto e seguro.

Somente com os componentes ZConnection e ZQuery é possível executar qualquer comando SQL, usando a biblioteca ZeosLib, conforme demonstrado no trecho de programa abaixo.

```
1.  function TFrmTeste.Exemplo1: boolean;
2.  var Query: TZQuery;
3.  begin
4.      Query := TZQuery.Create(NIL);
5.      try
6.        Query.Connection := ZConnection;
7.        Query.SQL.Text := 'select * from
    nometabela';
8.        Query.Open;
9.        Result := not Query.Eof;
10.     finally
11.       Query.Close;
12.       Query.Free;
13.   end;
14. end;
```

Este código serve somente como exemplo de comandos que podem ser executados, sem nenhuma outra pretensão.

Explicações:

A *linha 2* declara uma variável do tipo TZQuery. Vale mencionar aqui que esta classe implementa métodos de acesso a dados para leitura e gravação, independente de qual seja o SGBD.

Na *linha 4* está sendo criado uma instância da classe TZQuery

Na *linha 6* ligamos o componente recém cria-

do ao componente "ZConnection"

Na *linha 7* informamos o comando SQL a ser executado. Vale salientar que poderia ser qualquer comando SQL válido para o banco de dados selecionado na conexão.

Na *linha 8* optamos pelo método "Open" para indicar que o comando SQL informado anteriormente deverá retornar uma ou mais "tuplas" como resposta. Caso contrário, usaríamos o método "ExecSQL"

Na *linha 11* encerra-se a consulta ao banco.

Na *linha 12* é feita a liberação de memória.

UM PEQUENO EXEMPLO

Tendo como objetivo mostrar como é simples programar usando a ZeosLib, elaborei um pequeno projeto em "Delphi 7" fazendo acesso a um banco de dados "Firebird versão 1.5". O banco de dados utilizado foi o "EMPLOYEE.FDB", que é instalado por padrão no subdiretório de exemplos do Firebird.

Esta aplicação permite a navegação na tabela "country" usando o componente DBNavigator. Possui um DBGrid para exibir os dados cadastrados e permitir as operações básicas de atualização (Alterar, Apagar, Incluir). Além disso, foi implementada também a possibilidade de se filtrar os dados a serem apresentados no DBGrid, através da cláusula "where" do comando SQL, conforme visto na **figura 1**.

Somente dois componentes da biblioteca ZeosLib foram utilizados: **TZConnection** e **TZQuery**. O primeiro é para permitir a conexão com o banco e o segundo é para fazer o acesso a tabela "country" do banco de dados.

No evento "on create" do formulário foi feita a parametrização da conexão com o banco, conforme os dados abaixo:

```
Protocol := 'firebird-1.5';
Database:= '10.1.1.28:\DirTeste\Firebird_
1_5\examples\EMPLOYEE.FDB';
Password := 'masterkey';
User := 'sysdba';
```

O protocolo selecionado foi escolhido para que fosse feito acesso ao banco de dados Firebird versão 1.5.

Observem que na propriedade "Database", utilizei a nomenclatura "endereço_IP:Arquivo_do_banco_de_dados". O endereço IP do servidor pode ser substituído pelo nome da máquina servidora, desde que a rede esteja configurada corretamente para resolução de nomes.

Já no evento "on show" do formulário, foi estabelecida a conexão com o banco de dados através da atribuição "zConn.Connected := True", e logo em seguida foi dada a instrução para a ZeosLib executar o comando SQL "*select * from country*" que havia sido parametrizado no momento de construção do formulário.

O filtro foi implementado no evento "on click" de um CheckBox. Aqui é importante salientar que sempre antes de se alterar o comando SQL, devemos instruir a ZeosLib para encerrar o comando SQL atual, antes de alterá-lo e executá-lo novamente. Observem também a combinação das aspas para que o comando funcione corretamente.

No momento de encerrar a aplicação, fechamos a query e a conexão com o banco de dados.

A codificação seria algo bem semelhante ao código abaixo:

```
1.  procedure TfrmPais.FormCreate(Sender:
    TObject);
2.  begin
3.    with zConn do
4.      begin
5.      Protocol := 'firebird-1.5';
6.      Database := '10.1.1.28:\DirTeste\
    Firebird_1_5\examples\EMPLOYEE.FDB';
7.      Password := 'masterkey';
8.      User := 'sysdba';
9.      end;
10. end;
11.
12. procedure TfrmPais.FormShow(Sender:
    TObject);
13. begin
14.   zConn.Connected := True;
15.   zqPais.Active := True;
16. end;
```

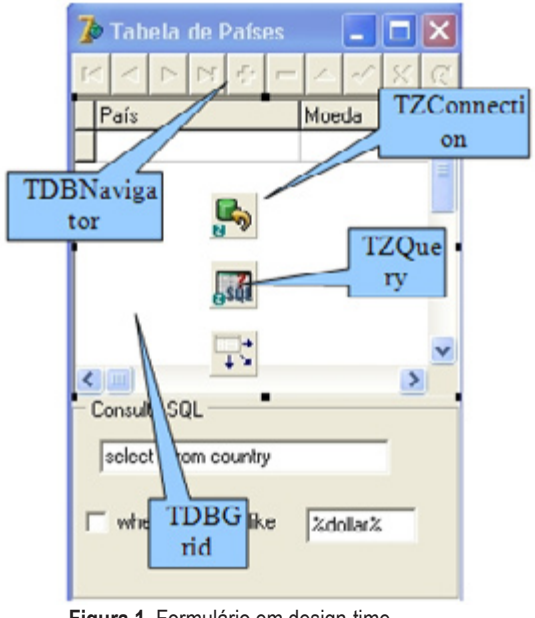


Figura 1. Formulário em design-time

```
17. procedure TfrmPais.
    FormCloseQuery(Sender: TObject; var
    CanClose: Boolean);
18. begin
19.   zqPais.Active := False;
20.   zConn.Connected := False;
21.   CanClose := True;
22. end;
23.
24. procedure TfrmPais.
    cbFiltrarClick(Sender: TObject);
25. begin
26.   zqPais.Active := False;
27.   zqPais.SQL.Clear;
28.   zqPais.SQL.Text := edtSQL.Text;
29.   if cbFiltrar.Checked then
30.     zqPais.SQL.Append(' where currency
    like ''''+edtMoeda.Text+'''');
31.   zqPais.Active := True;
32. end;
```

Figura 2. Aplicação em tempo de execução

A **figura 2** mostra a aplicação em tempo de execução.

CONCLUSÃO

ZeosLib é um projeto *freeware* e *open-source* que permite às aplicações desenvolvidas em Delphi, Kylix, C++ Builder e FPK/Lazarus acessar banco de dados de forma transparente e profissional.

O projeto ZeosLib é desenvolvido em Borland Delphi, portanto utilizando a linguagem de programação Object Pascal e as bibliotecas VCL/CLX.

É um projeto tecnicamente muito complexo e que sem sombra de dúvida atende de uma forma muito competente as maiores exigências de qualquer aplicação profissional, sendo ela comercial ou não.

O código fonte, muito bem escrito, propicia um grande aprendizado para os desenvolvedores que desejarem um aprimoramento na arte da programação orientada a objetos.

Em tempo de execução, a aplicação que utiliza a biblioteca ZeosLib irá ter um grande diferencial de mercado, pois é muito eficiente , com utilização de memória bem controlada e parametrizada, além de permitir que a mesma aplicação utilize diversos bancos de dados de forma totalmente transparente.

Avalie esse artigo

ÓTIMO **BOM** **RUIM**

Autor:

Mário Lúcio Gomes de Faria

Mini-curriculo

Formado em Ciência da Computação (UFMG), atua como Analista de Sistemas, Suporte de Redes e Software básico. Consultor SAP/R3 MM, gestor de Informática e professor. Tem larga experiência em Windows, Linux , Redes (TCP/IP) e Banco de Dados. Autor do software LPD32 disponível no site www.delphi32.com.

Calendário de Eventos

Data	Evento	Site
6 a 8 Abril/2006	2ª Escola Regional de Banco de Dados (ERBD) Passo Fundo - RS	http://inf.upf.br/erbd2006/
19 a 22 Abril/2006	FISL 7.0 - Porto Alegre - RS	http://fisl.softwarelivre.org/7.0/
17 a 20 Julho/2006	XXVI Congresso da Sociedade Brasileira de Computação Campo Grande - MS	http://www.ledes.net/pantaneiro_pg/sites/sbc/
29/Maio a 02/Junho 2006	24° Simpósio Brasileiro de Redes de Computadores Curitiba - PR	http://www.sbrc2006.arauc.br/
29/Maio a 02/Junho 2006	VII Workshop de Testes e Tolerância a Falhas (WTF) Curitiba - PR	http://www.sbc.org.br/index.php?language=1&subject=60&content=news&id=3695
29/Julho/2006	3º Firebird Developers Day Piracicaba - SP	http://www.FirebirdDevelopersDay.com.br

Se você sabe de algum evento focado em Banco de Dados ou em Software Livre que não esteja listado aqui, envie-nos um email com os dados do evento para que possamos incluí-lo na próxima edição e no calendário do site.