

P rojeto de Banco de Dados Parte VI: Implementação

Vinicius Lourenço de Sousa

Nas cinco edições anteriores, aprendemos a projetar um banco de dados relacional. Conhecemos os modelos lógico e físico, as técnicas de modelagem e a normalização e desnortinalização, entre outros procedimentos. Nesta edição, implementaremos o banco de dados de acordo com a documentação (diagramas, dicionários de dados) gerada nas fases anteriores.

Observe que, diferentemente das fases anteriores, as preocupações na fase de implementação se concentram no nível mais baixo. Nesta fase, por exemplo, será necessário analisarmos tipos de dados, tamanho de campos, uso de *constraints*, planejamento de índices, criação de *procedures* e *triggers*, bem como o SGBD a ser usado.

Este artigo não tem por objetivo discutir qual SGBD é o melhor. Usaremos o Firebird 1.0 simplesmente porque ele é *free*, leve, estável e popular. Também não desejo criar aqui um tutorial sobre o Firebird. Desse modo, é recomendável que o leitor esteja familiarizado com esse SGBD e tenha bons conhecimentos sobre SQL.

O artigo mostrará apenas os trechos mais relevantes do código SQL. Para obter o *script* completo, com todos os recursos implementados e o dicionário de dados, faça o download do código no site referente a este artigo.

Criação do banco de dados

Para a criação do banco de dados, utilizaremos o aplicativo **IBConsole**. Siga estes passos:

1) Inicie o **IBConsole**.

2) Você será solicitado a informar o *login*; o usuário e senha padrão são, respectivamente, *SYSDBA* e *masterkey*.

3) Clique com o botão inverso do mouse no item *Databases* e escolha a opção *Create Database*.

4) Na caixa de diálogo, preencha a opção *Alias* com o nome “*livraria*”; na opção *Filename*, indique qual o

caminho e o nome do arquivo que conterá o banco de dados (*c:\Livraria.gdb*) (Figura 1).

5) Clique no botão **OK** para criar o banco de dados. Os objetos do banco serão criados através do *Interactive SQL*, que é acessado através do menu **Tools->Interactive SQL**.

Nota: existem diversas ferramentas gráficas para manipulação de bancos de dados InterBase/Firebird. Uma das mais utilizadas é o **IBExpert** (<http://www.ibexpert.com/>), que também possui uma versão gratuita (Figura 2)

Criação de tabelas

O próximo passo é criar as tabelas “básicas”, ou seja, aquelas que não possuem chaves estrangeiras e que não dependem de outras tabelas para existir. Sendo assim, as primeiras tabelas a serem criadas são: *ASSUNTO*, *AUTOR*, *CLIENTE* e *EDITORA* (Listagem 1). Observações:

- Os nomes das tabelas e dos campos seguem o mesmo padrão do modelo físico;
- Os tipos de dados e as informações sobre o tamanho do campo podem ser definidos tanto no modelo físico como durante a implementação do banco. Neste artigo, escolhemos a segunda opção;

The image shows two windows from the IBExpert application. The left window, labeled F1, is the 'Create Database' dialog. It has fields for 'Server' (Local Server), 'Alias' (livraria), 'File(s)', and 'Options'. The 'File(s)' section contains a table with one row: 'Filename(s)' c:\Livraria.gdb and 'Size (Pages)' 4096. The right window, labeled F2, is the main application window showing the 'ASSINATURA' database. It displays tables like 'ASSINATURA' (with 1 row), 'PARCEIROS' (with 1 row), and 'JavMagazine' (with 242 rows). A detailed view of the 'JavMagazine' table is shown, listing columns: CODIGO (INTEGER NOT NULL), TITULO (VARCHAR(100)), AUTOR (VARCHAR(100)), EDITORA (VARCHAR(100)), CIDADE (VARCHAR(100)), UF (VARCHAR(2)), PAIS (VARCHAR(100)), and CEP (VARCHAR(10)). Below the table, the 'Interactive SQL' window shows a query: 'select us,cidade,count(cidade) from USUARIO group by us,cidade'.

L1

- Observe o comando `CONSTRAINT "nome_da_constraint" PRIMARY KEY("nome_do_campo")` nas linhas 5, 12, 19 e 26. Essa declaração informa que o campo passado como parâmetro de `primary key` será a chave primária da tabela.

Agora que temos as tabelas ‘básicas’, podemos criar as tabelas com chaves estrangeiras. Veja um exemplo parcial na **Listagem 2**. Observe a implementação de chaves estrangeiras nas linhas 4, 20, 28 e 29.

Implementando os relacionamentos

Na linguagem SQL, os relacionamentos são implementados por meio da criação de *constraints* de chaves estrangeiras. Essas *constraints* garantem o que chamamos de **integridade referencial**, que consiste em um conjunto de regras que mantêm o relacionamento íntegro (por exemplo: a tabela filho tem que fazer referência a um registro válido da tabela pai; não é possível apagar um registro da tabela pai que tenha ocorrências na tabela filho etc).

Quando existe qualquer tentativa de violação de uma *constraint*, o SGBD gera uma exceção. Essa exceção pode ser capturada pela aplicação cliente e, nesse caso, ela deve realizar o devido tratamento, emitindo uma mensagem amigável para o usuário.

Veja um exemplo de criação de *constraint* de chave estrangeira:

```
ALTER TABLE "ENDERECO"
ADD CONSTRAINT "FK3_CLIENTE"
FOREIGN KEY ("COD_CLIENTE")
REFERENCES "CLIENTE" ("CODIGO")
ON DELETE CASCADE
```

Nesse exemplo, o nome da *constraint* é `FK3_CLIENTE`. Ela implementa o relacionamento entre as tabelas *Endereco* (filho) e *Cliente* (pai). A cláusula `ON DELETE CASCADE` significa que, quando um registro da tabela pai for excluído, todos os endereços que estiverem relacionados a esse cliente também serão excluídos automaticamente. Também podemos usar a cláusula `ON UPDATE CASCADE` – nessa cláusula, quando o valor da chave primária na tabela pai for alterado, o SGBD atualizará automaticamente o valor da chave estrangeira nos registros filhos. Se as opções `ON DELETE/UPDATE` não

```
1 CREATE TABLE "ASSUNTO"
2 (
3   "CODIGO"      INTEGER NOT NULL,
4   "ASSUNTO"     VARCHAR(50) NOT NULL,
5   CONSTRAINT    "PK_ASSUNTO" PRIMARY KEY ("CODIGO")
6 );
7
8 CREATE TABLE "AUTOR"
9 (
10  "CODIGO"      INTEGER NOT NULL,
11  "NOME"        VARCHAR(80) NOT NULL,
12  CONSTRAINT    "PK_AUTOR" PRIMARY KEY ("CODIGO")
13 );
14
15 CREATE TABLE "CLIENTE"
16 (
17  "CODIGO"      INTEGER NOT NULL,
18  "NOME"        VARCHAR(80) NOT NULL,
19  CONSTRAINT    "PK_CLIENTE" PRIMARY KEY ("CODIGO")
20 );
21
22 CREATE TABLE "EDITORIA"
23 (
24  "CODIGO"      INTEGER NOT NULL,
25  "NOME"        VARCHAR(80) NOT NULL,
26  CONSTRAINT    "PK_EDITORIA" PRIMARY KEY ("CODIGO")
27 );
```

L2

```
1 CREATE TABLE "ENDERECO"
2 (
3   "CODIGO"      INTEGER NOT NULL,
4   "COD_CLIENTE" INTEGER NOT NULL,
5   "RUA"         VARCHAR(50) NOT NULL,
6   "NUMERO"      INTEGER,
7   "BAIRRO"      VARCHAR(30) NOT NULL,
8   "CIDADE"      VARCHAR(30) NOT NULL,
9   "CEP"          CHAR(8) NOT NULL,
10  "ESTADO"      CHAR(2) NOT NULL,
11  "TIPO"        CHAR(1) NOTNULL,
12  "COMPLEMENTO" VARCHAR(20),
13  CONSTRAINT    "PK_ENDERECO" PRIMARY KEY ("CODIGO")
14 );
15
16 CREATE TABLE "LIVRO"
17 (
18  "ISBN"        INTEGER NOT NULL,
19  "NOME"        VARCHAR(80) NOT NULL,
20  "COD_AUTOR"   INTEGER NOT NULL,
21  "FG_ALUGUEL" INTEGER,
22  "PRECO_RENOVACAO_ALUGUEL" DECIMAL(5, 2),
23  "FG_IMPORTADO" INTEGER,
24  "QUANTIDADE"  INTEGER NOT NULL,
25  "PRECO_VENDA" DECIMAL(5, 2),
26  "PRECO_ALUGUEL" DECIMAL(5, 2),
27  "ANO_PUBLICACAO" INTEGER NOT NULL,
28  "COD_EDITORA" INTEGER NOT NULL,
29  "COD_ASSUNTO" INTEGER NOT NULL,
30  CONSTRAINT    "PK_LIVRO" PRIMARY KEY ("ISBN")
31 );
```

forem utilizadas, ocorrerá um erro quando se tentar apagar/alterar o registro na tabela pai.

A **Listagem 3** mostra a criação de alguns relacionamentos através do comando `ALTER TABLE`. No primeiro exemplo, o campo `Cod_assunto` da tabela *Livro* faz referência ao campo `Codigo` da tabela *Assunto*, implementando o relacionamento entre essas tabelas.

Também podemos implementar auto-relacionamentos através de *constraints*. Por exemplo, note que na **Listagem 3** a *constraint* `FK1_PESSOA_FISICA` faz referência à própria tabela *Pessoa_fisica*. Isso significa que o campo `Cod_cliente_titular` não pode receber um valor diferente da chave primária dos clientes já cadastrados.

Também podemos adicionar o comando *constraint foreign key* diretamente ao

```

1. ALTER TABLE "LIVRO"
   ADD CONSTRAINT "FK1_ASSUNTO"
2. FOREIGN KEY ("COD_ASSUNTO")
3. REFERENCES "ASSUNTO" ("CÓDIGO");
4. ALTER TABLE "PESSOA_FÍSICA"
   ADD CONSTRAINT "FK1_PESSOA_FÍSICA"
   FOREIGN KEY ("COD_CLIENTE_TITULAR")
   REFERENCES PESSOA_FÍSICA ("COD_CLIENTE");

```

```

1. ALTER TABLE "ITEM_COMPRA"
   ADD CONSTRAINT "CK1_ITEM_COMPRA" CHECK(QTD_COMPRA > 0);
2. ALTER TABLE "LIVRO" ADD CONSTRAINT "CK2_LIVRO" CHECK(QUANTIDADE >= 0);
3. ALTER TABLE "LIVRO" ADD CONSTRAINT "CK3_LIVRO" CHECK(PRECO_VENDA >= 0);
4. ALTER TABLE "ALUGA"
   ADD CONSTRAINT "CK1_ALUGA"
   CHECK(PERÍODO_ALUGUEL > 0 AND PERÍODO_ALUGUEL <= 14);

```

```

1. CREATE EXCEPTION "EX_NUMERO_DEPENDENTE" 'O titular já possui 3 dependentes.';
2. CREATE TRIGGER "TR_VERIFICA_NUMERO_DEPENDENTE01" FOR "PESSOA_FÍSICA"
3. ACTIVE BEFORE INSERT POSITION 0 AS
4. DECLARE VARIABLE vTOTAL INTEGER;
5. BEGIN
6.   SELECT COUNT(*) FROM PESSOA_FÍSICA
7.   WHERE COD_CLIENTE_TITULAR = NEW.COD_CLIENTE_TITULAR
8.   INTO :vTOTAL;
9.   IF (vTOTAL = 3) THEN
10.    EXCEPTION EX_NUMERO_DEPENDENTE;
11. END;

```

```

1. CREATE EXCEPTION "EX_NUMERO_ITEM_ALUGA" 'Cota de livros por aluguel excedida.';
2. CREATE TRIGGER "TR_VERIFICA_NUMERO_ITENS" FOR "ITEM_ALUGA"
3. ACTIVE BEFORE INSERT POSITION 0 AS
4. DECLARE VARIABLE vTOTAL INTEGER;
5. BEGIN
6.   SELECT COUNT(*) FROM ITEM_ALUGA
7.   WHERE COD_ALUGA = NEW.COD_ALUGA
8.   INTO :vTOTAL;
9.   IF (vTOTAL = 3) THEN
10.    EXCEPTION EX_NUMERO_ITEM_ALUGA;
11. END;

```

comando CREATE TABLE, conforme mostra o exemplo a seguir:

```

CREATE TABLE "PESSOA_JURÍDICA"
(
  "COD_CLIENTE"      INTEGER NOT NULL,
  "TP_PESSOA_JURÍDICA" CHAR(1),
  "CNPJ"              CHAR(14) NOT NULL,
  CONSTRAINT "PK_PESSOA_JURÍDICA"
  PRIMARY KEY ("COD_CLIENTE"),
  CONSTRAINT "FK1_CLIENTE" FOREIGN KEY
  ("COD_CLIENTE")
  REFERENCES CLIENTE ("CÓDIGO")
  ON DELETE CASCADE;
)

```

Nota: observe o uso da cláusula NOT NULL na linha 29 da Listagem 2. Ela indica que o relacionamento entre Livro e Assunto é obrigatório, ou seja, não é possível cadastrar um livro sem

seu respectivo assunto. Quando o relacionamento não for obrigatório, basta omitir a cláusula NOT NULL na chave estrangeira. O que define se um relacionamento é ou não obrigatório é sua cardinalidade, levantada durante a fase de projeto.

Implementando as regras de negócio

Até aqui, já temos a estrutura principal (tabelas e relacionamentos) construída. O próximo passo é implementar as regras de negócio no banco de dados. Neste contexto, as 'regras de negócio' incluem, dentre outras operações, os requisitos levantados com o cliente, as restrições de integridade, as cardinalidades dos atributos

L3 e obrigatoriedade de alguns relacionamentos. Essas regras já foram levantadas e documentadas nas fases anteriores.

Geralmente, neste momento, costuma-se discutir quais regras serão controladas no SGBD e quais regras serão controladas no código da aplicação. Não entrarei nesse debate, já que ele demandaria um artigo separado; contudo, posso afirmar que, na maior parte dos casos, implementar as regras básicas usando os recursos do SGBD é boa prática.

O Firebird (e a maioria dos bancos relacionais) oferece três tipos de recursos para implementação de restrições:

1. Constraints de chave primária e chave estrangeira: já vimos esse mecanismo nos tópicos anteriores; com essas constraints, as regras de unicidade da chave primária e as regras de integridade dos relacionamentos são garantidas.

2. Constraints de campos: permitem definir validações para cada campo de uma tabela. O Firebird oferece as seguintes constraints de campo:

- **NOT NULL:** o preenchimento do campo se torna obrigatório;
- **UNIQUE:** assim como a constraint de chave primária, UNIQUE garante a exclusividade do campo na tabela, ou seja, nenhum valor poderá se repetir no campo que tiver essa constraint;
- **CHECK:** define uma regra booleana para o campo. O valor inserido é testado junto a condição CHECK. Se a expressão retornar verdadeiro, o valor será aceito.

Veja um exemplo de uso de constraints de campo:

```

CREATE TABLE "FUNCIONARIO"
(
  "NOME"      VARCHAR(60) NOT NULL,
  "CPF"        CHAR(11) UNIQUE NOT NULL,
  "SALARIO"    NUMERIC(15,2)
               CHECK(SALARIO >= 0),
  "DEPTO"      INTEGER CHECK (DEPTO > 1 AND
                               DEPTO <= 10)
);

```

A Listagem 4 exibe algumas regras do projeto que foram implementadas com o uso de constraints de campo. Observe especialmente a cláusula check da linha 4: conforme definido no levantamento de requisitos, o período de aluguel não pode exceder duas semanas.

A **Tabela 1** exibe um resumo das regras de negócio que foram implementadas no banco.

3. Controle procedural: algumas regras só podem ser validadas através de rotinas procedurais. Atualmente, os SGBDs oferecem a possibilidade de armazenar e executar rotinas por meio de objetos conhecidos como *triggers* e *stored procedures*.

Em alguns casos, o administrador pode preferir não codificar as validações mais complexas no banco de dados e deixar essa responsabilidade para a equipe de desenvolvimento. Normalmente, ele decide isso com base no tipo de aplicação que está sendo construída e na política de desenvolvimento adotada pela equipe.

No projeto da livraria, todas as validações procedurais foram escritas dentro do próprio banco de dados. Vejamos alguns exemplos:

Triggers: consistem em rotinas associadas a eventos que podem ocorrer em uma tabela. Veja a seguir algumas regras que foram implementadas com triggers:

■ **O cliente titular pode ter, no máximo, 3 dependentes.**

No Firebird, não temos como limitar o número de relacionamentos pela *constraint* de chave estrangeira. Para resolver isso, podemos implementar duas triggers (uma BEFORE INSERT e uma BEFORE UPDATE) para impedir que a aplicação associe mais de três dependentes a um titular. Veja o código da trigger BEFORE INSERT na **Listagem 5**.

A expressão SQL da linha 6 à 8 retorna o número atual de dependentes do titular em questão. Esse resultado é armazenado na variável *vTOTAL*. Caso o valor desta variável seja igual a “3”, será disparada a exceção *EX_NUMERO_DEPENDENTE*.

■ **Os clientes só podem alugar três livros por vez.**

Para esta restrição foi criada a trigger *TR_VERIFICA_NUMERO_ITENS* (**Listagem 6**). Note que os itens alugados são registrados na tabela *Item_aluga*. Antes de cada inserção, a expressão SQL da linha 6 à 8 calcula a quantidade de itens inseridos no aluguel em questão. Se essa quantidade já estiver no limite (três), será disparada uma exceção.

Stored Procedures: consistem em procedimentos armazenados no SGBD. A diferença em relação a *trigger* reside no fato de que uma *stored procedure* não permanece associada a nenhum objeto ou evento do banco de dados.

Uma *stored procedure* não oferece a mesma segurança que uma *trigger*. Ao implementar uma restrição com triggers, o administrador pode ficar seguro de que a regra não será violada – como a *trigger* permanece associada a um evento de tabela, nenhuma aplicação poderá fazer inserções, alterações ou exclusões na tabela sem que a *trigger* referente seja executada. No caso de *stored procedures*, a aplicação pode ‘burlar’ a regra simplesmente deixando de usar a *stored procedure* para manipular a tabela. Uma forma de minimizar esse risco é impedir que os usuários tenham poderes para

executar operações de inserção, alteração ou exclusão na tabela (e conceder apenas poder de execução na *stored procedure*).

Responder à questão ‘quando usar uma trigger no lugar de uma *stored procedure*’ não é o foco deste artigo; desse modo, vou apenas detalhar as regras que foram implementadas com *stored procedures*.

O primeiro exemplo refere-se à renovação do aluguel. Quando o locatário renovar o aluguel, o sistema deverá:

1) Replicar os dados do aluguel atual para um novo aluguel, ou seja, o registro na tabela *Aluguel* e os registros relacionados na tabela *Item_aluguel* deverão ser copiados e reinseridos;

2) No momento da inserção, recuperar o valor do aluguel de cada livro do campo *Preco_renovacao_aluguel*, da tabela *Livro*. Para fins de simplificação, o sistema não cadastrá um *flag* indicando que o novo aluguel é proveniente de uma renovação;

3) Fechar o aluguel original.

Dessa forma, podemos codificar uma *stored procedure* para receber o código do aluguel como parâmetro e realizar todo esse procedimento automaticamente. Com isso, obtemos os seguintes benefícios:

a) Minimizamos o risco de o desenvolvedor errar: no momento da renovação, a aplicação deverá se preocupar apenas em executar a *procedure* e passar os parâmetros necessários – o desenvolvedor não precisará codificar a ‘lógica’ do processo de renovação;

b) Diminuimos o tráfego na rede: a aplicação não precisará enviar vários comandos SQL para o servidor.

Nome	Objeto	Tipo	Objetivo	T1
CK1_LIVRO	Tabela LIVRO	CHECK CONSTRAINT	Verifica se o preço de renovação de aluguel é maior ou igual a zero	
CK1_ALUGA	Tabela ALUGA	CHECK CONSTRAINT	Verifica se o período de aluguel é menor que duas semanas	
TR_VERIFICA_NUMERO_DEPENDENTE	Tabela PESSOA_FISICA	TRIGGER BEFORE INSERT/UPDATE	Garante que cada cliente titular tenha, no máximo, três dependentes	
TR_CLIENTE_TITULAR_SI_MESMO	Tabela PESSOA_FISICA	TRIGGER BEFORE INSERT/UPDATE	O cliente não pode ser titular de si mesmo	
TR_VALIDAR_ITEM_RESERVA	Tabela ITEM_RESERVA	TRIGGER BEFORE INSERT	Verifica se um livro já está reservado	
SP_RENOVAR_ALUGUEL	-	STORED PROCEDURE	Implementa a renovação de um aluguel	

Algumas validações implementadas no banco de dados

```

1. CREATE PROCEDURE "SP_RENOVAR_ALUGUEL"
2. (
3. COD_ALUGUEL INTEGER,
4. COD_CLIENTE INTEGER,
5. VALOR_MULTA DECIMAL(6, 2),
6. PERIODO_ALUGUEL INTEGER
7. )
8. AS
9. DECLARE VARIABLE vCODIGO_ALUGUEL INTEGER;
10. DECLARE VARIABLE vISBN_LIVRO INTEGER;
11. DECLARE VARIABLE vPRECO_RENOVACAO_ALUGUEL DECIMAL(5,2);
12. BEGIN
13. /* Encerra o aluguel que será renovado */
14. UPDATE ALUGA SET
15. DT_DEVOLUCAO= CURRENT_DATE,
16. VR_MULTA= :VALOR_MULTA
17. WHERE CODIGO=:COD_ALUGUEL;

18. /* Cria um novo aluguel */
19. INSERT INTO ALUGA(COD_CLIENTE, DT_HORA, PERIODO_ALUGUEL)
20. VALUES(:COD_CLIENTE, CURRENT_TIMESTAMP, :PERIODO_ALUGUEL);

21. SELECT MAX(CODIGO) FROM ALUGA INTO :vCODIGO_ALUGUEL;

22. /* Obtém todos os itens do aluguel que será renovado para
23. incluí-los como itens do novo aluguel */
24. FOR SELECT ISBN_LIVRO FROM ITEM_ALUGA
25. WHERE COD_ALUGA = :COD_ALUGUEL
26. INTO :vISBN_LIVRO
27. DO
28. BEGIN

29. /* Para cada livro, obtém o preço de renovação para
30. colocar como preço do aluguel renovado */
31. SELECT PRECO_RENOVACAO_ALUGUEL FROM LIVRO
32. WHERE ISBN = :vISBN_LIVRO
33. INTO :vPRECO_RENOVACAO_ALUGUEL;

34. /* Insere os livros como itens do novo aluguel */
35. INSERT INTO ITEM_ALUGA(COD_ALUGA, ISBN_LIVRO, PRECO)
36. VALUES(:vCODIGO_ALUGUEL, :vISBN_LIVRO, :vPRECO_RENOVACAO_ALUGUEL);
37. END
38. END

```

```

CREATE PROCEDURE "SP_INCLUI_CLIENTE"
(
    "NOME" VARCHAR(80),
    "DT_NASCIMENTO" DATE,
    "IDENTIDADE" VARCHAR(9),
    "CPF" CHAR(11),
    "COD_CLIENTE_TITULAR" INTEGER,
    "TP_PESSOA_JURIDICA" INTEGER,
    "CNPJ" CHAR(14),
    "TIPO_PESSOA" CHAR(1)
)
AS
DECLARE VARIABLE vCOD_CLIENTE INTEGER;

BEGIN
    INSERT INTO CLIENTE (NOME) VALUES(:NOME);

    SELECT MAX(CODIGO) FROM CLIENTE INTO :vcod_cliente;

    IF (:TIPO_PESSOA='F') THEN
        BEGIN
            INSERT INTO PESSOA_FISICA (COD_CLIENTE, DT_NASCIMENTO, IDENTIDADE, CPF,
                                      COD_CLIENTE_TITULAR)
            VALUES(:vcod_cliente, :DT_NASCIMENTO, :IDENTIDADE, :CPF,
                   :COD_CLIENTE_TITULAR);
        END

    IF (:TIPO_PESSOA='J') THEN
        BEGIN
            INSERT INTO PESSOA_JURIDICA (COD_CLIENTE, TP_PESSOA_JURIDICA, CNPJ)
            VALUES(:vcod_cliente, :TP_PESSOA_JURIDICA, :CNPJ);
        END
    END

```

L7

Por exemplo, no caso do aluguel de três livros, a aplicação precisaria enviar quatro comandos SQL (um para recadastrar o registro da tabela *Aluguel* e três para a tabela *Item_Aluguel*). Com a *procedure*, a aplicação enviará apenas um comando pela rede (o de execução da *procedure*).

Veja o código completo da *procedure* na **Listagem 7**. Vamos analisá-lo:

Linhas 9 a 11: declaração das variáveis que serão utilizadas na *procedure*;

Linha 14: o aluguel atual é encerrado. Na linha 19, é inserido um novo aluguel;

Linha 24: o laço FOR percorre todos os itens previamente alugados, na tabela *Item_aluga*;

Linha 31: recupera o preço de renovação para cada livro em questão;

Linha 35: inclui novamente cada livro alugado na tabela *Item_aluguel*;

Em seguida, analisaremos uma segunda regra que pede a criação de uma *stored procedure*. No projeto, criamos uma especialização através das tabelas *CLIENTE*, *PESSOA_FISICA* e *PESSOA_JURIDICA*.

A integridade dessa especialização deve ser garantida por intermédio dos seguintes controles:

1) Um cliente não pode ser pessoa física e jurídica ao mesmo tempo.

2) Não é possível cadastrar um registro na tabela *Pessoa_fisica* nem na tabela *Pessoa_juridica* sem que haja um equivalente na tabela *Cliente*, e vice-versa (o relacionamento entre a tabela *Cliente* e uma das tabelas especializadas é obrigatório).

Criaremos uma *stored procedure* que receberá todas as informações do cliente (nome, CPF, CNPJ, endereço etc) e as distribuirá entre as tabelas correspondentes, de acordo com o tipo de cliente informado (físico ou jurídico). A aplicação só poderá cadastrar os clientes através dessa *procedure*.

O código da *procedure* está disponível na **Listagem 8**.

Implementando as *surrogate keys*

Conforme mostrado na edição anterior, após a aplicação do recurso de minimização de chaves, as tabelas passaram a ter apenas um campo auto-incremento como

L9
 CREATE TRIGGER "TR_ALUGA" FOR "ALUGA"
 ACTIVE BEFORE INSERT POSITION 0
 AS
 BEGIN
 NEW.CODIGO = GEN_ID(GEN_ALUGA, 1);
 END

chave primária. No processo de criação das tabelas, esse campo foi declarado *integer*, não aceitando valor nulo (*not null*). No entanto, até o momento não há nada que efetivamente indique que o campo será auto-incremento.

No Firebird, a melhor forma de implementar a geração de números auto-incremento é através do objeto *Generator*. Esse objeto garante que nenhum número será repetido, mesmo que a transação corrente seja cancelada – um requisito fundamental para chaves do tipo *surrogate*.

Para associar um *generator* a uma chave primária, siga estes passos:

- 1) Crie o objeto *generator*.
- 2) Implemente uma trigger *before insert* para a tabela que possui a chave auto-incremento. Nessa trigger, o *generator* será incrementado, e seu valor será atribuído à chave primária.

Para criar um *generator*, utilize o comando a seguir:

```
CREATE GENERATOR "nome_do_generator";
```

O passo 2 está exemplificado na **Listagem 9**. A função GEN_ID incrementa o *generator* de acordo com a quantidade informada no segundo parâmetro.

Esse procedimento deverá ser repetido para todas as tabelas que possuem chaves *surrogate*.

Planejamento de índices

Um dos últimos passos para a implementação do banco de dados é o planejamento e a criação dos índices. Não entrarei nos detalhes dessa tarefa, pois ela pertence à área de otimização (*tuning*) do banco de dados - tema bastante complexo e que exigiria, no mínimo, um novo artigo.

Para resumir, podemos afirmar que os campos mais utilizados para consulta devem fazer parte de algum índice. Veja um exemplo de criação de índices no Firebird:

```
CREATE INDEX IDX_NOME ON
CLIENTE(NOME);
```

Para ter acesso à criação de todos os índices levantados para o nosso banco de dados, consulte o script SQL disponível para download.

Dicionário de dados

Cada fase da construção do banco de dados (projeto lógico, projeto físico e implementação) gera uma documentação conhecida como dicionário de dados. O dicionário de dados é a fonte de consulta para referência. Nele, devemos descrever todos os objetos, relacionamentos, comportamentos, restrições e observações levantadas durante cada fase.

O dicionário de dados criado na fase de implementação é constituído, em sua maior parte, de metadados dos objetos implementados no banco. Praticamente todos os SGBDs já fornecem ferramentas para gerar relatórios de metadados. O dicionário de dados completo de nosso exemplo está disponível para download no site.

Nota: metadado é uma informação que descreve um tipo de dado.

Conclusão

Chegamos ao final da série. Neste ponto, temos os principais diagramas do projeto do banco, bem como o script SQL para sua implementação.

Um dos objetivos deste artigo foi mostrar a você a importância de um projeto de banco de dados. Observe como um projeto bem documentado e bem construído lhe oferece uma visão mais ampla e concisa do funcionamento do banco, o que poderá ajudá-lo no caso de qualquer alteração futura na aplicação.

Espero ter sido bem-sucedido em minha tentativa de torná-lo apto a iniciar seus projetos. Caso você tenha alguma dificuldade, sinta-se à vontade para me escrever por e-mail. Um abraço e até a próxima! ■

**Faça download e comente
essa matéria em:**

www.sqlmagazine.com.br/sql7

S

A

Vinicius Lourenço de Sousa

(vsouza@dba.com.br) é membro da equipe editorial da SQL Magazine, analista de sistemas, projetista e desenvolvedor Delphi/Java para projetos Web e Off-Line que utilizam bancos de dados Interbase, Firebird, Oracle 9i e DB2/AS400 na DBA Engenharia de Sistemas. É Pós-Graduado em Análise, Projeto e Gerência de Sistemas pela PUC-RJ e possui certificação BrainBench em Delphi e RDBMS.

Treinamento em Software Livre para profissionais de TI

DEXTRA

www.dextra.com.br/treinamento
treinamento@dextra.com.br
Fone: (19) 3256-8644

- ✓ PostgreSQL
- ✓ CVS
- ✓ JBoss
- ✓ Linux
- ✓ PHP

10% de Desconto

para leitores SQL Magazine

* Informar código **sqlmag**
 Promoção válida até 31/12/2003