

DELPHI

Delphi

© Wendel Wagner Martins Nogueira
Rua General Osório nº 2551
CEP – 79824-040
Dourados - MS
Telefone (067) 421-1532 • (067) 422-5122 ramal 238
Informatica@cpao.embrapa.br

Índice Analítico

INTRODUÇÃO AO MODELO CLIENTE/SERVIDOR.....	1
MUDANÇAS DE PARADIGMAS	1
<i>Paradigma Computacional.....</i>	<i>1</i>
<i>Paradigma do Negócio.....</i>	<i>3</i>
EVOLUÇÃO DA ARQUITETURA	4
<i>Arquitetura Time-Sharing.....</i>	<i>4</i>
<i>Arquitetura Resource-Sharing.....</i>	<i>5</i>
<i>Arquitetura Cliente/Servidor</i>	<i>6</i>
PRIMEIRA GERAÇÃO CLIENTE/SERVIDOR	8
SEGUNDA GERAÇÃO CLIENTE/SERVIDOR	8
SGDB - SISTEMAS GERENCIADORES DE BANCO DE DADOS	10
MODELOS DE BANCO DE DADOS.....	10
<i>Sistema de Gerenciamento de Arquivos</i>	<i>11</i>
<i>Banco de Dados Hierárquico</i>	<i>11</i>
<i>Banco de Dados de Rede</i>	<i>12</i>
<i>Banco de Dados Relacional.....</i>	<i>13</i>
BANCOS DE DADOS RELACIONAIS	15
CLASSIFICAÇÃO	15
<i>Corporativos</i>	<i>16</i>
<i>Departamentais</i>	<i>16</i>
<i>Locais ou Móveis.....</i>	<i>16</i>
MODELAGEM DE DADOS.....	17
<i>Normalização.....</i>	<i>17</i>
<i>Propagação de chaves primárias</i>	<i>22</i>
<i>Ferramentas.....</i>	<i>24</i>
<i>Criação da Base de Dados</i>	<i>24</i>
<i>Utilizando Interbase Windows ISQL.....</i>	<i>24</i>
LINGUAGEM SQL	26
<i>Categorias da Linguagem SQL</i>	<i>27</i>
<i>Utilizando o Windows ISQL para definir o Banco de Dados.....</i>	<i>27</i>
<i>Utilizando o Windows ISQL para acessar o Banco de Dados</i>	<i>27</i>
CONSISTÊNCIA E INTEGRIDADE DOS DADOS	29
<i>Integridade Referencial</i>	<i>29</i>
<i>Domínio dos dados.....</i>	<i>30</i>
<i>Regras de Negócio.....</i>	<i>30</i>
<i>Utilizando o Windows ISQL para definir integridades e consistências</i>	<i>31</i>
<i>Utilizando o Windows ISQL para testar as consistências.....</i>	<i>31</i>
<i>Distribuição da Consistência e Integridade dos Dados</i>	<i>33</i>
SQL EXPLORER	34
<i>CRIAÇÃO DE ALIAS</i>	<i>34</i>
<i>VISUALIZAÇÃO E EDIÇÃO DE DADOS</i>	<i>35</i>

DEFINIÇÃO DE NOVOS ELEMENTOS	37
DEFINIÇÃO DE DICIONÁRIOS DE DADOS	37
<i>Criação de um novo Dicionário</i>	37
<i>Importação das definições do Banco de Dados</i>	38
<i>Definição das propriedades dos Attribute Sets</i>	38
<i>Utilização do Dicionário no Delphi</i>	39
TRABALHANDO COM BANCOS DE DADOS RELACIONAIS.....	41
COMPONENTES DA ARQUITETURA CLIENTE/SERVIDOR	41
CONEXÕES E CONTEXTOS	42
<i>Conexões e Contextos no Delphi</i>	43
CURSORES E RESULT SETS	45
<i>Cursorres e Result Sets no Delphi</i>	46
TRANSAÇÕES	46
<i>Transações no Delphi</i>	47
CONCORRÊNCIA	51
<i>Tipos de travamentos (locks)</i>	51
<i>Níveis de isolamento</i>	52
<i>Optimistic Lock</i>	53
<i>Concorrência no Delphi</i>	54
PROJETANDO APLICAÇÕES CLIENTE/SERVIDOR.....	61
ESTRUTURA DE UMA APLICAÇÃO.....	61
<i>Apresentação</i>	61
<i>Lógica do Negócio</i>	61
<i>Gerenciamento de Dados</i>	62
VANTAGENS DA ORGANIZAÇÃO DA APLICAÇÃO EM CAMADAS	62
ESTRUTURA DE UMA APLICAÇÃO DELPHI	63
<i>Componentes visuais</i>	63
<i>Componentes de Acesso à base de dados</i>	64
<i>Componente de ligação</i>	64
CONSTRUINDO APLICAÇÕES CLIENTE/SERVIDOR	65
UTILIZANDO DATA MODULES	65
COMPONENTE TDATABASE.....	66
ESCOLHENDO ENTRE TTABLE E TQUERY	67
<i>Abertura</i>	67
<i>Filtros</i>	67
<i>Transações</i>	68
<i>Número de Tabelas Acessadas</i>	68
TRABALHANDO COM O TQUERY	69
UTILIZANDO CACHED UPDATES	70
UTILIZANDO O COMPONENTE TUPDATESQL.....	72
GRAVAÇÃO LINHA A LINHA OU EM BATCH.....	74
TRABALHANDO COM O TTABLE.....	78
FILTRANDO REGISTROS	81
QBE NA MESMA TELA DE MANUTENÇÃO	82
CONTROLANDO OS ESTADOS DA TELA.....	85
TELA DE CONSULTA ESPECÍFICA.....	92
<i>Tela de Consulta</i>	92
<i>Tela de Manutenção</i>	98
<i>Recursos de LookUp</i>	102
<i>Buscando Registros de outras TabSheets</i>	105
<i>Controlando Transações</i>	106
<i>Mantendo o Result Set da Consulta</i>	108
CONTROLANDO TRANSAÇÕES MASTER/DETAIL.....	111

TELA DE CONSULTA.....	111
TELA DE MANUTENÇÃO.....	116
<i>Lógica visual</i>	119
<i>Controle visual da Transação</i>	123
<i>Transação - Lógica de negócio</i>	124
<i>Calculando o Valor Total do Pedido</i>	125
<i>Regras de Negócio</i>	126
APÊNDICE A (TRATAMENTO DE EXCEÇÕES).....	127
<i>Aplicações robustas</i>	127
<i>Principais exceções</i>	127
<i>Blocos de finalização</i>	127
<i>Gerando exceções</i>	127
<i>Erros de Bancos de dados</i>	127
<i>Aplicações robustas</i>	127

Introdução ao Modelo Cliente/Servidor

Esse capítulo mostra a evolução na arquitetura de computadores e a tendência a ambientes de aplicações cada vez mais distribuídos.

O termo Cliente/Servidor foi inicialmente aplicado para a arquitetura de software que descrevia o processamento entre dois programas. Nesse contexto, a aplicação cliente requisitava um serviço que era então executado pelo programa servidor. Entretanto, esse termo ainda não distinguia se o programa cliente e o programa servidor estavam sendo executados em uma mesma máquina ou em máquinas diferentes. Hoje, quando se fala em cliente/servidor, está se referindo a um processamento cooperativo distribuído, onde o relacionamento entre clientes e servidores são relacionamentos entre componentes tanto de software quanto de hardware. Portanto, estaremos interessados na arquitetura cliente/servidor envolvendo duas ou mais máquinas em um processo cooperativo para executar a aplicação.

Mudanças de Paradigmas

Paradigma Computacional

A visão tradicional da computação era centralizada na figura do computador, que concentrava todos os serviços e recursos fornecidos aos usuários. O acesso a esses computadores era feito diretamente pelo usuário através de teclados e monitores. Em alguns casos, essa máquina era um equipamento poderoso capaz de atender vários usuários simultaneamente (Mainframe). Ou então, eram pequenos computadores isolados capazes de atender um único usuário de cada vez.

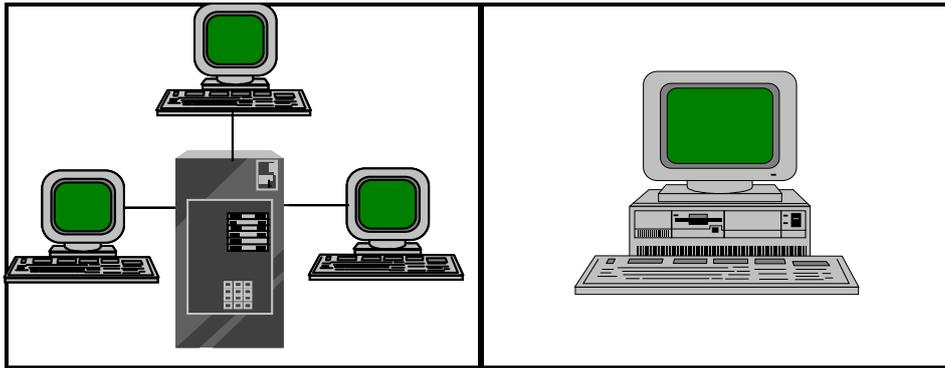


Fig. 1.1: Arquitetura de Mainframe à esquerda e um computador PC isolado à direita. Ambos simbolizam a visão computacional tradicional, onde o foco é o computador.

Para que essas máquinas pudessem acompanhar a crescente demanda de novos recursos e serviços seria necessário expandir seus recursos de memória e processamento. No caso de se utilizar apenas uma máquina servindo vários usuários (Mainframe), seria difícil expandir ainda mais seus recursos para suportar as novas demandas tecnológicas, principalmente a interface gráfica. Por outro lado, através de várias máquinas isoladas, os serviços teriam que ser replicados e cada máquina incrementada para suportar tal processamento. Além disso, essa alternativa não provê uma maneira de comunicação entre os usuários.

Para atender a necessidade crescente do usuário de novos serviços e recursos, surgiu-se a necessidade de interligar essas máquinas, para que juntas pudessem fornecer um número maior de benefícios. Assim começaram a aparecer ambientes de redes que permitiam a distribuição de recursos e serviços em locais diferentes, aumentando a capacidade do usuário final. Com isso as máquinas que tinham que ser equipamentos caros e poderosos podiam ser aos poucos substituídas por várias máquinas menores e de custo mais baixo, mas que através de um processo cooperativo conseguiam prover funcionalidade maior.

Com o número cada vez maior de computadores interligados, esses deixaram de ser o foco do ambiente computacional e toda a atenção e importância foram destinadas às redes, elementos responsáveis em reunir os recursos e serviços de várias máquinas e disponibilizá-los aos usuários de forma transparente e eficaz.

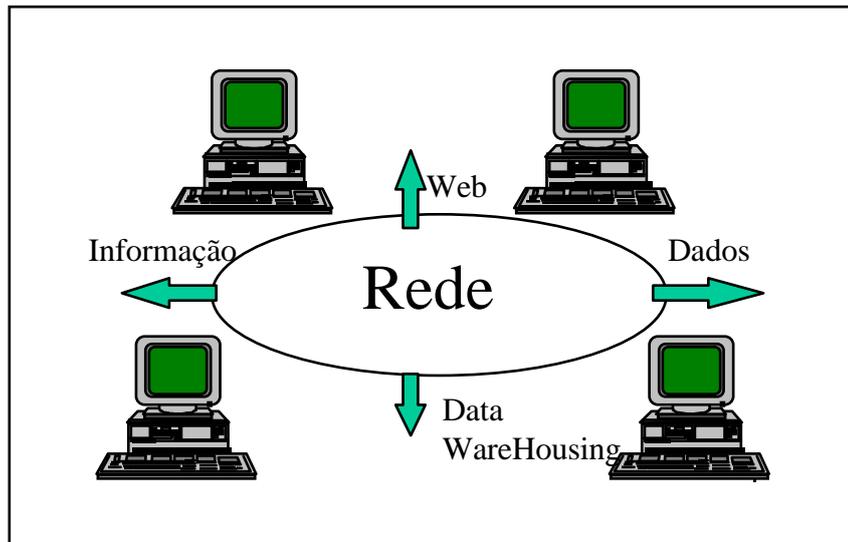


Fig 1.2: As redes interligam os computadores fornecendo serviços e recursos de forma escalonada e transparente para os usuários finais.

As redes permitem o crescimento do ambiente computacional de forma flexível e escalonada, possibilitando a disponibilidade de novos recursos e serviços ao usuário final com uma certa transparência. Hoje, serviços novos como Web, DataWareHousing e outros podem ser mais facilmente agregados ao parque computacional e disponibilizados a vários usuários já pertencentes a rede. Nesse modelo, o computador se torna apenas um meio de acesso a infinidade de serviços e recursos oferecidos por toda a rede.

Paradigma do Negócio

Do ponto de vista do negócio, as organizações estão sendo afetadas por grandes mudanças no mercado. As empresas precisam se preparar para a globalização da economia e serem mais rápidas na tomada de decisão tornando-se mais competitivas.

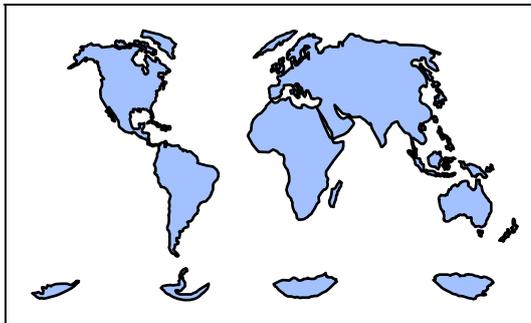


Fig 1.3: Globalização da economia. As empresas atuam mundialmente, não havendo mais limites geográficos para a competitividade.

Uma atenção especial deve ser dada aos clientes que exigem produtos cada vez mais específicos e personalizados. Para suportar essa pressão competitiva do mercado, os gerentes de negócio devem estar munidos de informações e ferramentas que os permitam tomar decisões de forma rápida e precisa. Informações específicas e fundamentais devem ser extraídas da base de dados corporativa para darem suporte a essas decisões. Portanto, a velocidade que a informação trafega pela empresa é muito importante e um ambiente capaz de prover essa funcionalidade se torna realmente necessário.

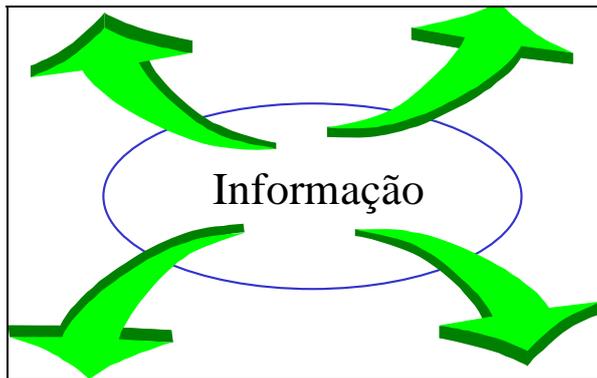


Fig. 1.4: Descentralização da informação.

Com o objetivo de se tornarem mais competitivas, as empresas têm procurado uma forma de se organizar melhor promovendo um conjunto de mudanças organizacionais e estruturais em toda a corporação. Técnicas como “Reengenharia” são normalmente aplicadas, com o objetivo de reorganizar a empresa e seus processos de negócio. Essas organizações do negócio são muitas vezes acompanhadas de uma reestruturação dos sistemas de informática e de todo o ambiente computacional. É preciso estruturar um novo ambiente descentralizado e eficiente que proporcione a funcionalidade e a velocidade desejada para viabilizar a corporação em um mercado cada vez mais abrangente e competitivo.

Evolução da Arquitetura

O objetivo da arquitetura cliente/servidor é proporcionar um ambiente capaz de compartilhar os recursos e serviços de várias máquinas interligadas e prover uma maneira cooperativa de executar as aplicações dos usuários finais.

A seguir serão mostradas algumas arquiteturas para explicar a forte tendência a ambientes distribuídos cooperativos.

Arquitetura Time-Sharing

Essa arquitetura é baseada em um processamento centralizado. Uma máquina, chamada de hospedeiro, é responsável por rodar todos os programas e gerenciar todos os recursos. O tempo de processamento é compartilhado pelos programas, simulando uma execução em paralelo. Os usuários têm acesso a esses serviços e recursos através de terminais conectados localmente ou remotamente. Esse terminais não possuem nenhuma capacidade de processamento e consistem basicamente de uma tela, um teclado e do hardware necessário para se comunicar com o hospedeiro. Essa arquitetura permite o compartilhamento da base de dados da aplicação tornando a informação disponível de qualquer terminal. Entretanto, com o surgimento de novas necessidades, como a interface gráfica e outros serviços que necessitam cada vez mais de processamento, esse modelo começou-se a tornar economicamente e fisicamente inviável já que todo o processamento é realizado em uma única máquina. Os exemplos mais conhecidos dessa arquitetura são os sistemas em Mainframes e alguns sistemas em UNIX.

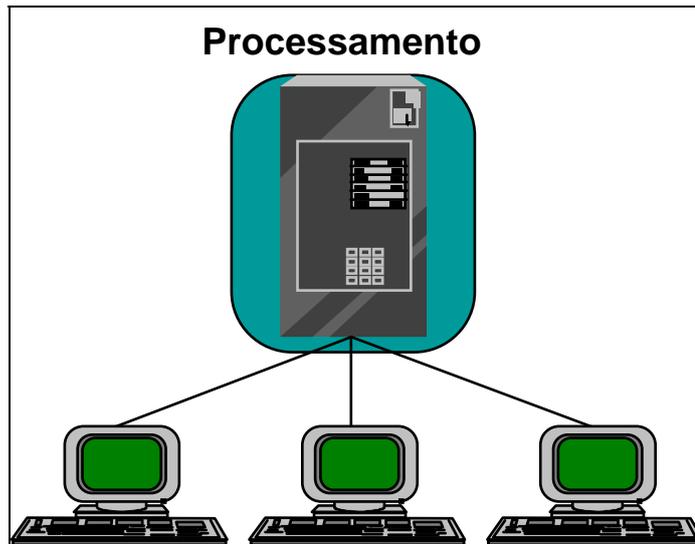


Fig 1.5. Arquitetura Time-Sharing

Arquitetura Resource-Sharing

Essa arquitetura consiste de vários computadores (estações de trabalho) interligados, sendo cada um capaz de realizar seu próprio processamento. Alguns desses computadores são responsáveis em compartilhar e gerenciar recursos tais como impressora, disco, etc (servidores de rede).

Entretanto, a rede não é utilizada para proporcionar um processamento cooperativo entre as máquinas. Todo o processamento da aplicação é ainda feito por uma única máquina, havendo apenas o compartilhamento de recursos. Através dessa arquitetura é possível compartilhar a base de dados da aplicação, permitindo o acesso por várias pessoas simultaneamente. Mas como todo o processamento dos dados é realizado em cada máquina, a necessidade de um volume maior de informações torna inviável o tráfego de informações pela rede. Para resolver esse problema seria necessário que a máquina responsável em armazenar os dados fizesse um processamento local capaz de enviar uma quantidade menor de dados para a máquina que está processando a aplicação.

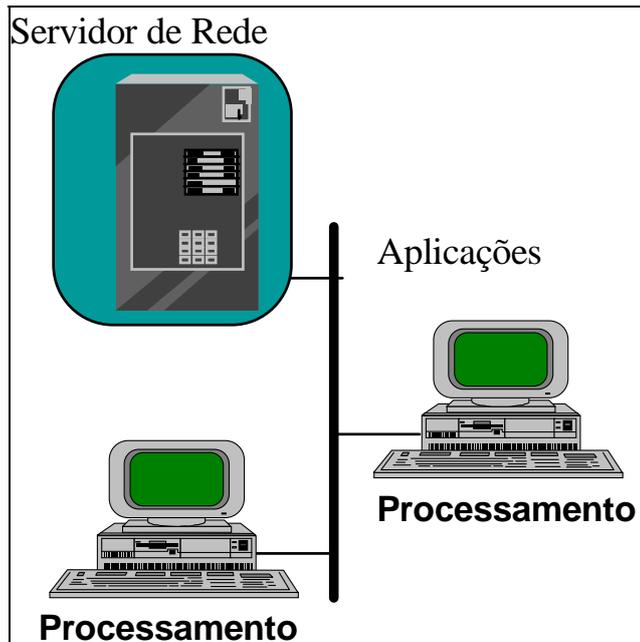


Fig. 1.6: Arquitetura Resource-Sharing.

Arquitetura Cliente/Servidor

Essa arquitetura também consiste de vários computadores, cada um com seu próprio processamento, interligados em rede. A diferença básica para a arquitetura Resource-Sharing é que aqui já começa a haver um processamento distribuído cooperativo. Parte do processamento, que era feito pela máquina da aplicação, é feito agora pela própria máquina responsável pelo armazenamento e distribuição da informação, diminuindo assim o tráfego de informações na rede. Portanto, pode-se e deve-se selecionar os dados que serão enviados para o usuário para uma melhor eficiência do ambiente.

Esse modelo já começa a retirar partes específicas de processamento das aplicações que eram executadas pelas máquinas clientes, centralizando-as nas máquinas de localização física mais adequada, garantindo assim uma melhor distribuição do processamento e utilização do ambiente. Através dessas especializações garante-se também um melhor gerenciamento e facilidade de manutenção dos serviços devido a sua concentração em um ou poucos locais físicos.

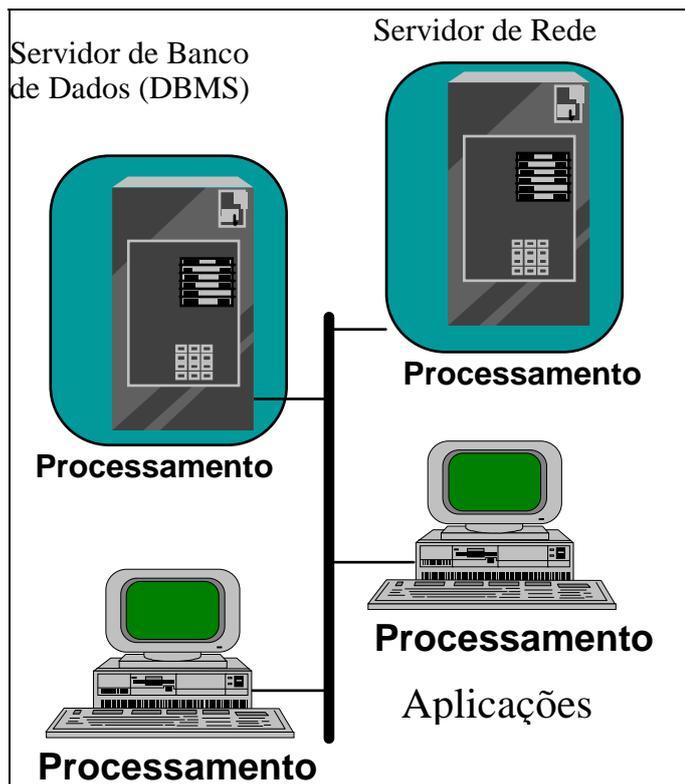


Fig. 1.7 Arquitetura Cliente/Servidor.

Podemos citar alguns benefícios que esta nova arquitetura traz para o ambiente computacional das empresas:

- Permite às corporações alavacarem a tecnologia de computadores “desktops”. Hoje, as estações de trabalho já possuem um poder computacional considerável, por um custo muito mais baixo, antes só disponível em “Mainframes”.
- Permite que o processamento seja feito mais próximo da origem dos dados reduzindo o tráfego na rede.
- Facilita a utilização de aplicações gráficas e multimídias. Isto permite a construção de aplicações que excedem as expectativas dos usuários proporcionando-lhes um real aumento de produtividade.
- Permite e encoraja a utilização de sistemas abertos, já que clientes e servidores podem rodar em diferentes hardwares e softwares, livrando as corporações de arquiteturas proprietárias.

Entretanto, essa arquitetura ainda não é perfeita. Algumas características necessárias para um completo processo distribuído ainda não foram observadas nesse modelo que ainda apresenta algumas deficiências:

- Se uma porção significativa da lógica da aplicação for movida para o servidor, esse se torna um “gargalo” assim como na arquitetura de “Mainframe”. Para resolver esse problema seria necessário uma melhor distribuição e gerenciamento do processamento da

lógica da aplicação. Isto dá origem a uma nova arquitetura chamada “Segunda Geração Cliente/Servidor”.

- O processo de construção de aplicações distribuídas é bem mais complexo que o desenvolvimento de aplicações não distribuídas devido ao maior número de parâmetros a serem definidos, ao desconhecimento da tecnologia e a falta de padrões e ferramentas que auxiliem essa ambientação. Portanto, muito tempo pode ser consumido no processo de definição e construção do ambiente de desenvolvimento. Esse tempo é muitas vezes subestimado pelas empresas devido ao desconhecimento e as falsas propagandas dos vendedores de ferramentas.

Primeira Geração Cliente/Servidor

Podemos definir cliente/servidor como uma arquitetura computacional que visa distribuir os recursos e o processamento de forma inteligente com o objetivo de otimizar o desempenho da rede e dos sistemas, maximizando a utilização dos recursos de cada máquina e fornecendo uma base sólida e flexível para a implantação de um número crescente de serviços.

Algumas implementações desse tipo já vêm sendo utilizadas em várias empresas e são conhecidas como a primeira geração cliente/servidor.

Para compartilhar recursos, como disco e impressora, são utilizados Servidores de Arquivos na rede. Estes são sistemas com a função de processar as requisições aos arquivos e impressoras e gerenciar seu acesso e distribuição.

Além disso, parte do processamento das aplicações também foi distribuído. Alguns serviços de manipulação e gerenciamento de dados foram retirados das aplicações e colocados em pontos centralizados conhecidos como “Servidores de Banco de Dados”, tornando o processamento dos dados mais próximo do seu local de armazenamento. Os sistemas que fornecem tais serviços foram chamados de “Sistemas Gerenciadores de Banco de Dados” - SGDB.

Basicamente, a primeira geração de cliente/servidor se caracteriza por essa distribuição do processamento da aplicação entre dois componentes: a estação de trabalho do usuário e o servidor de banco de dados. À medida que a arquitetura cliente/servidor evolui, novas partes da aplicação vão sendo distribuídas e novos elementos vão aparecendo no ambiente.

Segunda Geração Cliente/Servidor

Hoje, a tecnologia cliente/servidor já caminha para sua segunda geração. Essa geração explora mais o ambiente de rede e suas máquinas. Surgem novos servidores com a finalidade de retirar das estações de trabalho grande parte do processamento que elas realizam. Os principais elementos dessa nova arquitetura são os servidores de aplicação e os servidores “Web”.

Os servidores de aplicação são responsáveis por retirar o restante da camada de manipulação de dados que ainda havia na estação cliente. Além disso, tem o objetivo de concentrar a lógica de negócio, antes distribuída entre a estação cliente e o servidor de banco. Normalmente, esse trabalho não é feito por um único servidor de aplicação e sim por um

conjunto de servidores onde o processamento é balanceado através de elementos chamados “Middleware”. Desta forma resta para a estação cliente, o processamento da interface visual com o usuário, deixando-a mais leve, exigindo uma menor configuração e melhorando seu desempenho.

Os servidores “Web” tentam ir mais longe ainda, permitindo retirar das estações de trabalho até parte da lógica da interface visual, deixando-as responsáveis apenas por interpretar o código “HTML” enviado pelos servidores. Entretanto, com a utilização de componentes como Java e ActiveX, parte do processamento pode retornar à estação de trabalho.

Essas novas tecnologias trazem mais recursos, mas tornam o ambiente mais complexo e difícil de ser implementado. É preciso estar bem certo do que se pretende e não fazer uso de uma tecnologia mais nova sem a conhecê-la direito ou sem a real necessidade de utilizá-la.

SGDB - Sistemas Gerenciadores de Banco de Dados

Esse capítulo apresenta os sistemas gerenciadores de banco de dados, seus serviços, recursos e modelos.

Os SGBDs são sistemas responsáveis em armazenar os dados e fornecer serviços capazes de manipulá-los. Para armazenar os dados em um disco, os SGBDs possuem uma forma estruturada padrão de representar os registros e campos, fornecendo serviços que permitem a criação e alteração dessas definições de dados. Fornecem ainda, um mecanismo interno para manter os dados no disco e para saber onde está cada elemento em particular. Também são responsabilidades do SGBDs disponibilizar serviços para incluir, classificar, atualizar e eliminar os dados armazenados.

Pela forma com que os dados são armazenados e de como eles se relacionam podemos classificar os SGBDs através de vários modelos até chegarmos aos bancos de dados relacionais.

Modelos de Banco de Dados

Analisando os diversos modelos de banco de dados, pode-se notar uma evolução no relacionamento entre os dados, eliminando cada vez mais a redundância e proporcionando mais flexibilidade e portanto uma maior facilidade de manutenção das definições de dados. Para melhor entendermos os modelos vamos utilizar os dados das três guias de pedidos da fig. 2.1.



Fig. 2.1: Guia de Pedido.

Sistema de Gerenciamento de Arquivos

Esse é o único sistema que descreve como os dados são armazenados. Nesse modelo cada campo ou item de dado é armazenado seqüencialmente no disco em um único e grande arquivo. Para encontrar um item de dado é necessário verificar cada elemento desde o início do arquivo. A única vantagem desse método é sua simplicidade, já que seu formato muito se parece com o de um arquivo texto onde as palavras se encontram escritas em uma determinada seqüência. Entretanto, por causa dessa simplicidade, muitos dados têm que ser escritos repetidamente gerando uma enorme redundância que dificulta a manutenção e a integridade dos dados. Não há também nenhuma indicação de relacionamento entre os dados, e o programador precisa saber exatamente como os dados estão armazenados para poder acessá-los de forma consistente. Além disso, sua estrutura física é rígida, o que dificulta alterações na definições dos dados, gerando a necessidade de reconstruir todo o arquivo.

```
1 | 20/10/96 | Ana Maria Lima | 999.876.555-22 | Caneta | 10
10,00 | Lápis | 5 | 5,00 | 15,00 | 2 | 21/10/96 | Maria José |
111.111-22 | Caneta | 15 | 15,00 | Caderno | ...
```

Fig. 2.2: Sistema de Gerenciamento de Arquivos.

Banco de Dados Hierárquico

Nesse modelo, os dados são organizados em uma estrutura de árvore que se origina a partir de uma raiz. Essa estrutura identifica as relações “pai-filho” entre os vários itens do banco de dados, mostrando assim suas vantagens sobre o modelo de gerenciamento de arquivos. No

modelo hierárquico é possível definir relações “de-um-para-muitos” que facilita e acelera o processo de pesquisa dos dados. Para encontrar uma informação, não é mais necessário percorrer o arquivo inteiro. Basta examinar o item pedido, decompô-lo em componentes e descer pelos ramos necessários até encontrá-lo. Esse método, também facilita a inserção de novos dados, devido aos relacionamentos entre os campos serem feitos através de ponteiros. Para inserir um novo elemento basta alterar os ponteiros dos relacionamentos entre pais e filhos.

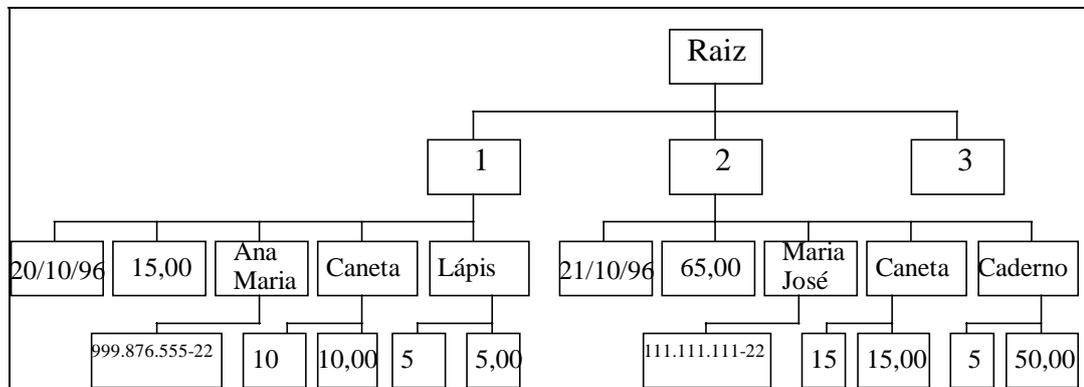


Fig. 2.3: Banco de Dados Hierárquico.

Entretanto, esse método ainda possui alguns problemas. Cada nível da árvore é inicialmente definido e qualquer alteração na estrutura desses níveis é uma tarefa difícil. Além disso, o problema da redundância de dados ainda não foi resolvido, já que esse modelo não implementa relações de “muitos-para-muitos”. No exemplo, podemos notar a repetição de algumas informações, o que dificulta a integridade e manutenção dos dados.

Banco de Dados de Rede

O modelo de rede descreve, conceitualmente, os banco de dados nos quais permitem relações de “muitos-para-muitos” entre os elementos de dados. Desta forma cada item possui um ponteiro para os itens com os quais se relaciona, eliminando assim a necessidade de qualquer tipo de redundância de dados. O grande problema desse modelo é a sua complexidade devido a flexibilidade existente em suas relações. Quando o volume de dados começa a crescer, os relacionamentos entre os itens de dados ficam cada vez mais complexos, tornando sua visualização e entendimento cada vez mais difíceis.

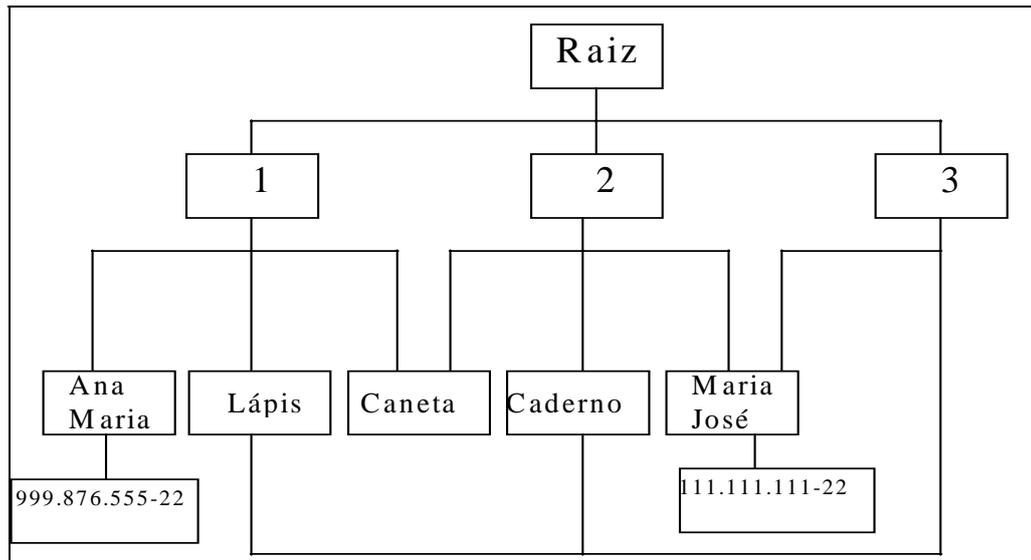


Fig. 2.4: Banco de Dados de Rede.

Banco de Dados Relacional

Em 1969, foi publicado pelo Dr. E. F. Codd, o primeiro artigo que definia um modelo com base no conceito matemático dos conjuntos relacionais. A partir desse artigo, o modelo relacional tem sido refinado, até que 1985 o próprio Dr. Codd lançou as “12 regras” que definiam um banco de dados relacional. Em 1990, foram publicadas as 333 regras que são subconjuntos e expansões das 12 regras originais.

O modelo relacional abandona o conceito de relações “pai-filho” feitas diretamente entre os dados e os organiza em conjuntos matemáticos lógicos de estrutura tabular. Nesse modelo, cada item de dado pertence a uma coluna da tabela e uma linha da tabela é composta por vários elementos diretamente relacionados.

As tabelas também se relacionam através de funções matemáticas como JOINS e UNIONS. Para fazer esse relacionamento parte dos dados, que identificam unicamente o registro da tabela, são repetidos dentro da outra tabela.

As vantagens desse método são sua simplicidade e flexibilidade nas definições das relações entre os vários itens de dados, já que não são feitos diretamente entre os dados e sim entre as tabelas. Entretanto, esse método não elimina por completo a redundância de dados, já que no mínimo os relacionamentos entre as tabelas são feitos através da repetição de parte dos dados. Além dessa redundância, fica a cargo do projetista do banco de dados se mais repetições de dados irão ou não fazer parte do modelo. O processo de fragmentação dos dados, a fim de serem organizados em subconjuntos (tabelas), é conhecido como normalização.

Pedido				Itens			
NumPed	Data	Valor	Cliente	NumPed	Produto	Quantid.	Valor
1	20/10/96	15,00	1	1	Caneta	10	10,00
2	21/10/96	65,00	2	1	Lápis	5	5,00
3	22/10/96	25,00	2	2	Caneta	15	15,00
				2	Caderno	5	50,00
				3	Lápis	15	15,00
				3	Caderno	1	10,00

Cliente		
Codigo	Nome	CPF
1	Ana Maria Lima	999.876.555-22
2	Maria José	111.111.111-22

Fig. 2.5: Banco de Dados Relacional.

Bancos de Dados Relacionais

Esse capítulo apresenta algumas características dos bancos de dados relacionais.

Os bancos de dados relacionais vêm se tornando um padrão no mercado, servindo como base de dados para a maioria dos sistemas das empresas. A cada dia, mais ferramentas são construídas para tirar proveito dessa tecnologia, fazendo surgir um número crescente de recursos e produtos a serem oferecidos para os usuários. Pode-se dizer então, que esta é uma tecnologia sólida e consistente e que irá acompanhar o mercado por um longo período de tempo. No entanto, é uma tecnologia que continua evoluindo com o objetivo de disponibilizar as informações para os usuários de maneira eficiente, viabilizando o negócio da corporação e descentralizando as tomadas de decisão.

Classificação

Quanto a capacidade, recursos e facilidade de uso, os banco de dados relacionais podem ser divididos em três categorias: corporativos, departamentais e locais. Em uma mesma empresa todas as categorias podem coexistir e cooperar entre si para juntas formarem uma poderosa base de dados distribuída.

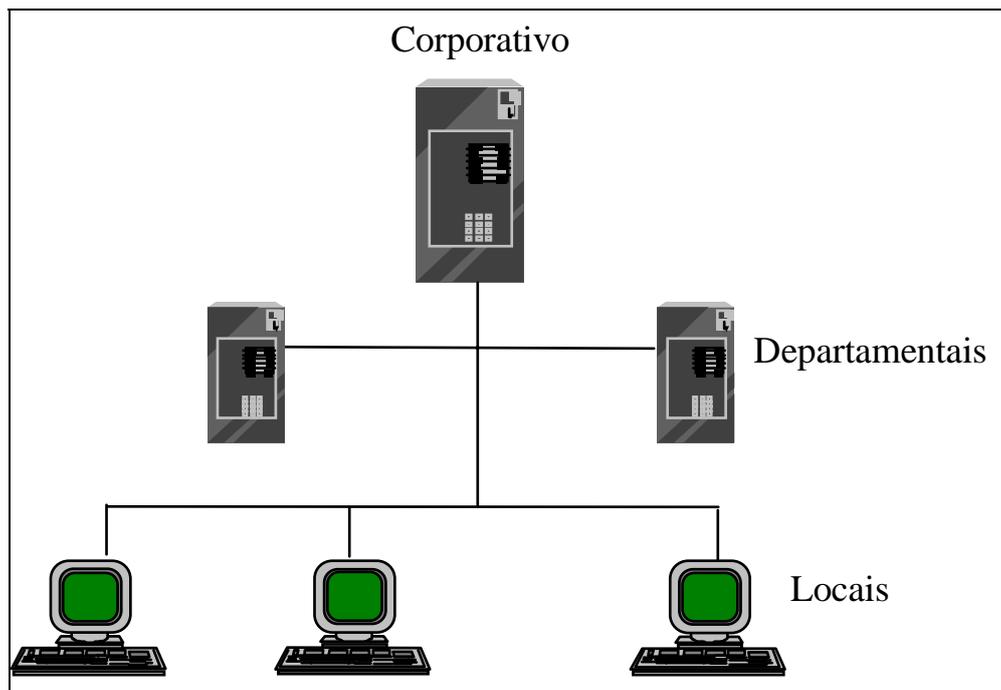


Fig 3.1: Classificação dos bancos de dados.

Corporativos

Sistema de banco de dados para atender toda a corporação. Os principais sistemas dessa categoria são: DB2, Oracle, Sybase, Informix e Ingres. Estes bancos devem ser capazes de armazenar e manipular de maneira eficiente grandes volumes de dados e permitir acesso rápido a um número elevado de usuários conectados ao mesmo tempo. Devido a sua enorme capacidade, os preços desses bancos são também mais elevados e normalmente necessitam de profissionais especializados para configurá-los e monitorá-los no dia a dia.

Departamentais

Sistemas de banco de dados capazes de atender as requisições de dados a nível de um departamento. Nesse patamar, existem bancos de dados mais baratos e mais fáceis de configurar, entre eles estão: Interbase, SQLServer, SQLBase.

A maioria dos fornecedores de bancos de dados da categoria corporativa estão também disponibilizando versões departamentais para disputar esse mercado, trazendo como vantagem a possibilidade de permanecer com o mesmo fornecedor em todos os níveis, tornando o ambiente mais integrado. Por outro lado, as principais vantagens dos bancos desenvolvidos especialmente para essa plataforma são: a facilidade de configuração, o preço dos produtos e a menor requisição de recursos de hardware.

Locais ou Móveis

Sistemas de banco de dados capazes de rodar na mesma máquina que a aplicação. Nessa categoria os principais requisitos são: ser um banco de dados bastante leve que gaste poucos recursos do sistema operacional e da máquina(memória, processamento, etc) e que não necessite de gerenciamento podendo rodar basicamente com sua configuração original.

Nessa categoria podemos citar os bancos Interbase, SQLBase e SQLAnywhere. Os fornecedores corporativos também estão descendo para esse nível. Porém, para manter os bancos de dados com os mesmos recursos tem sido difícil torná-los leves e fáceis suficientemente. Alguns fornecedores possuem até sistemas de banco de dados diferentes para cada plataforma.

Como esses bancos são instalados na mesma máquina é possível sua utilização em “notebooks”. Isso permite as empresas manterem a mesma tecnologia de banco de dados relacional em aplicações móveis, facilitando a integração com a base corporativa da empresa.

Deve-se notar entretanto, que não está se fazendo uso da arquitetura cliente/servidor nesse caso, já que a mesma máquina é utilizada para rodar o aplicativo e o sistema gerenciador de banco de dados. A principal vantagem dessa utilização sobre a utilização de base de dados de arquivos, é que a última não possui recursos suficientes para manter a integridade dos dados e das transações. Os sistemas gerenciadores de banco de dados são também menos factíveis a corrupção de dados e falhas do que os sistemas baseados em arquivos onde o gerenciamento é feito por cada aplicação. Outra vantagem é prover mais facilidade de integração com as bases departamentais e corporativas, por utilizarem a mesma tecnologia. Entretanto, os SGDBs possuem um custo adicional e exigem uma configuração de máquina melhor devido a um número maior de camadas que terão que ser percorridas para se chegar ao dado.

Modelagem de Dados

Modelagem de dados é o processo utilizado para definir a estrutura do banco de dados através da distribuição dos dados nas tabelas. Existem maneiras diferentes de se definir o mesmo conjunto de dados, e uma boa modelagem de dados pode facilitar bastante o desenvolvimento das aplicações. Vamos ressaltar aqui, dois pontos que devem ser observados na construção do modelo: o processo de normalização e a propagação de chaves primárias.

Normalização

Técnica de análise e organização de dados que visa determinar a melhor composição para uma estrutura de dados. Os principais objetivos dessa técnica são:

- Eliminar anomalias que dificultam as operações sobre os dados;
- Minimizar as redundâncias e os conseqüentes riscos de inconsistências;
- Reduzir e facilitar as manutenções.

Como exemplo, vamos utilizar um modelo, no qual todos os dados foram colocados em uma única tabela através dos seguintes campos:

NumPed, DataPed, ValorPed, NomeCliente, CPFCliente, NomeProduto1, PreçoProduto1, Quantidade1, ValorItem1, NomeProduto2, PreçoProduto2, Quantidade2, ValorItem2,..., NomeProduto5, PreçoProduto5, Quantidade5, ValorItem5
--

Primeira Forma Normal

Uma tabela está na primeira forma normal se não possuir grupos de atributos repetitivos.

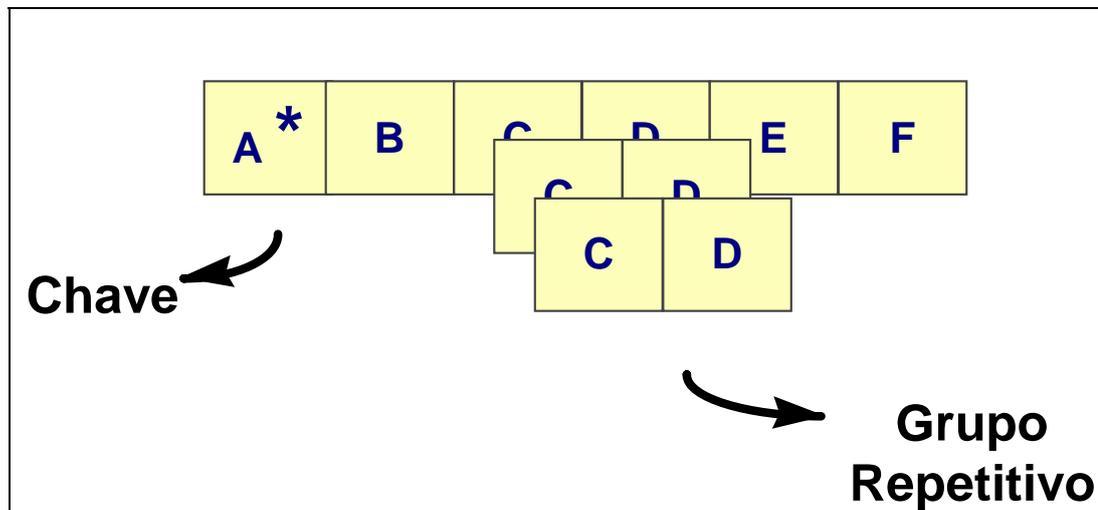


Fig. 3.2: Presença de grupos repetitivos em uma tabela relacional.

Para resolver esse problema, deve-se remover os grupos repetitivos para uma outra tabela. A chave primária dessa nova tabela pode ser formada através da chave primária da tabela original mais um conjunto de novos atributos.

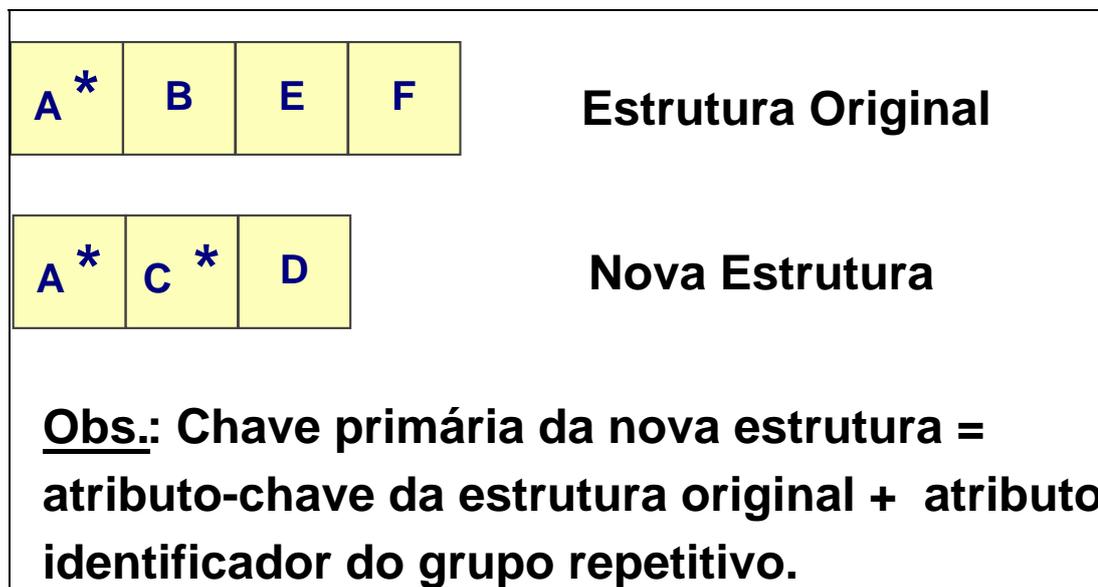


Fig. 3.3: Eliminação de grupos repetitivos em uma tabela.

Desta forma, elimina-se a deficiência do primeiro modelo que limitava o número de itens de produto que um pedido podia ter. Nessa nova estrutura não há limite para os itens, mas continua-se mantendo a relação que existia entre os itens e o pedido através da inclusão do número do pedido na nova tabela.

*NumPed DataPed ValorPed NomeCliente CPFCliente	*NumPed *NomeProduto PreçoProduto Quantidade ValorItem
---	--

Fig. 34: Exemplo na primeira forma normal.

Segunda Forma Normal

Uma tabela está na segunda forma normal quando sua chave primária não for composta ou quando todos os atributos “não chaves” forem funcionalmente dependentes da chave inteira e esta estiver na primeira forma normal. Um atributo “b” é dependente funcionalmente de “a” se dado o valor de “a”, o valor de “b” puder ser determinado.

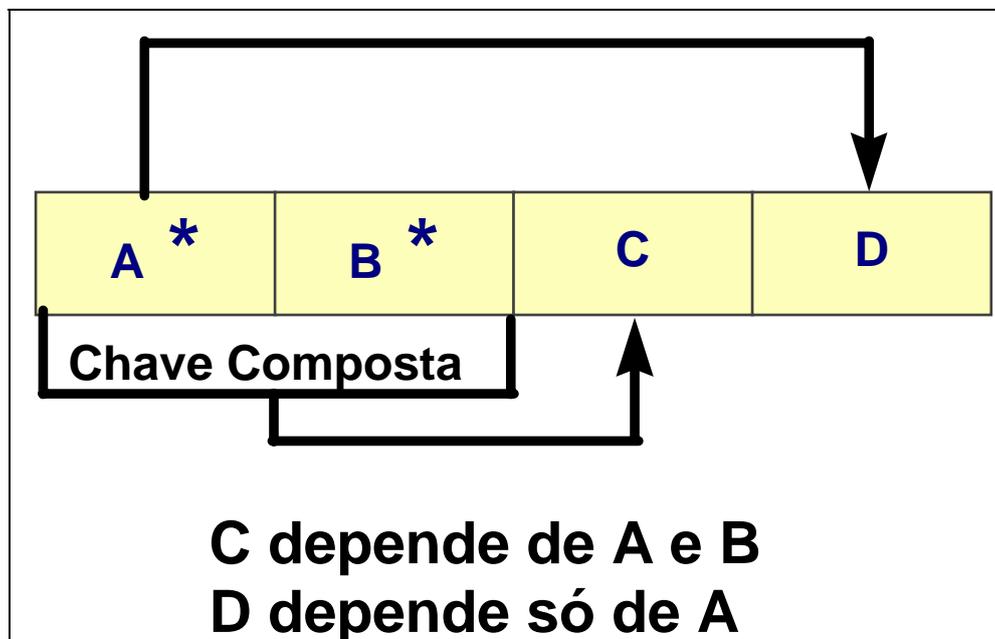


Fig. 3.5: Modelo na 1FN mas não em 2FN. Presença de atributos dependentes funcionalmente de apenas parte da chave primária.

Para resolver esse problema, basta remover os atributos que dependem apenas de parte da chave primária para uma nova tabela. A chave primária da nova tabela passa a ser então essa parte da chave primária da tabela original da qual o atributo dependia.

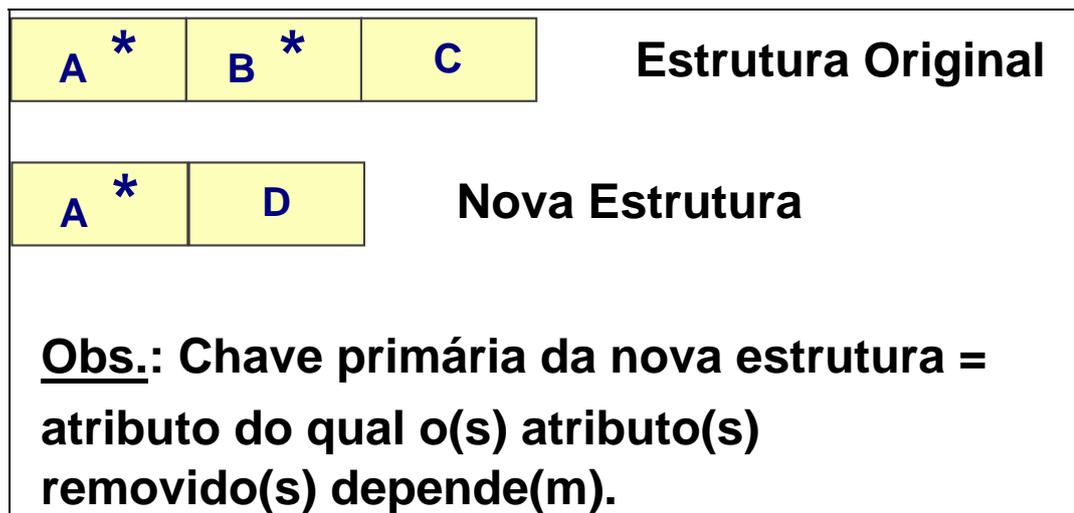


Fig 3.6: Restruturação do modelo na segunda forma normal

No exemplo de pedido, isso ocorre com a coluna PRECOPRODUTO, que depende apenas do NOMEPRODUTO e não depende de NUMPED. Dado o nome do produto é possível saber seu preço não importando o pedido que o produto pertença. Assim o preço do produto seria repetido em vários registros da estrutura desnecessariamente e dificultando a manutenção e consistência dos dados. Para evitar essa redundância, deve-se criar uma nova estrutura e a chave primária dessa nova tabela seria o NOMEPRODUTO, porém por questões de eficiência pode-se criar uma nova coluna CODPRODUTO de tamanho menor para fazer a ligação entre as tabelas. Além da eficiência, esse procedimento possibilita alterações futuras nos nomes dos produtos, já que esses não são mais usados como chave primária.

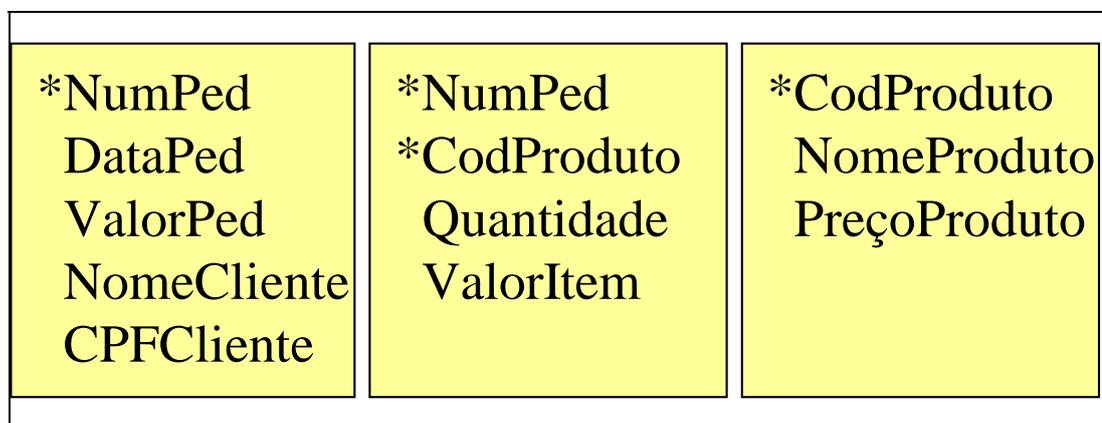


Fig. 3.7. Exemplo na segunda forma normal.

Terceira Forma Normal

Uma tabela está na terceira forma normal quando está na segunda forma normal e não possui nenhum atributo “não chave” dependendo funcionalmente de outro atributo “não chave”.

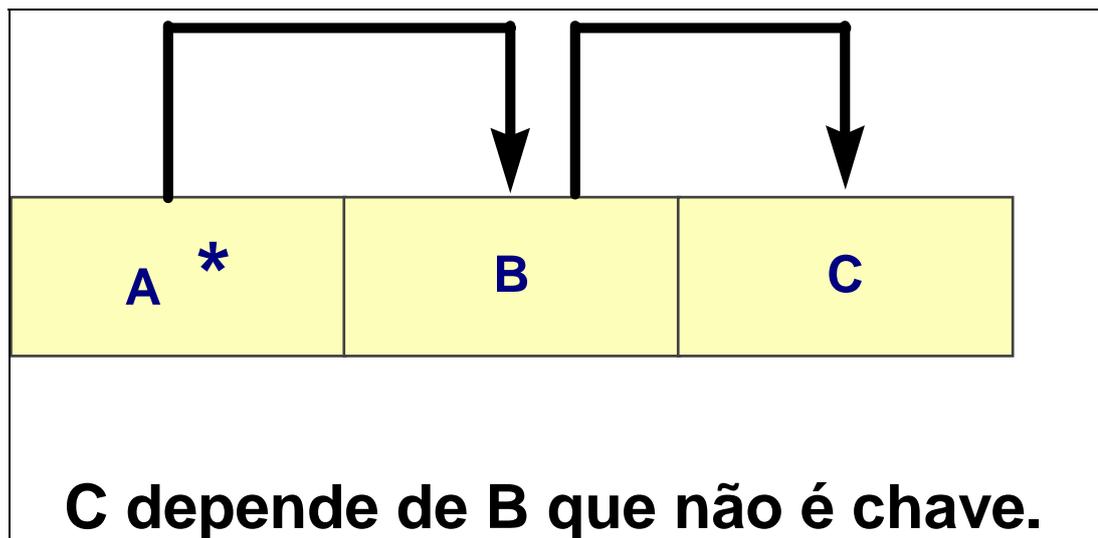


Fig. 3.8: Presença de atributos dependentes funcionalmente de outros atributos não-chave.

Para resolver esse problema, basta remover os atributos que dependem de outros atributos não chave para uma nova tabela. A chave primária da nova tabela passa a ser então o atributo não chave que possui dependentes.

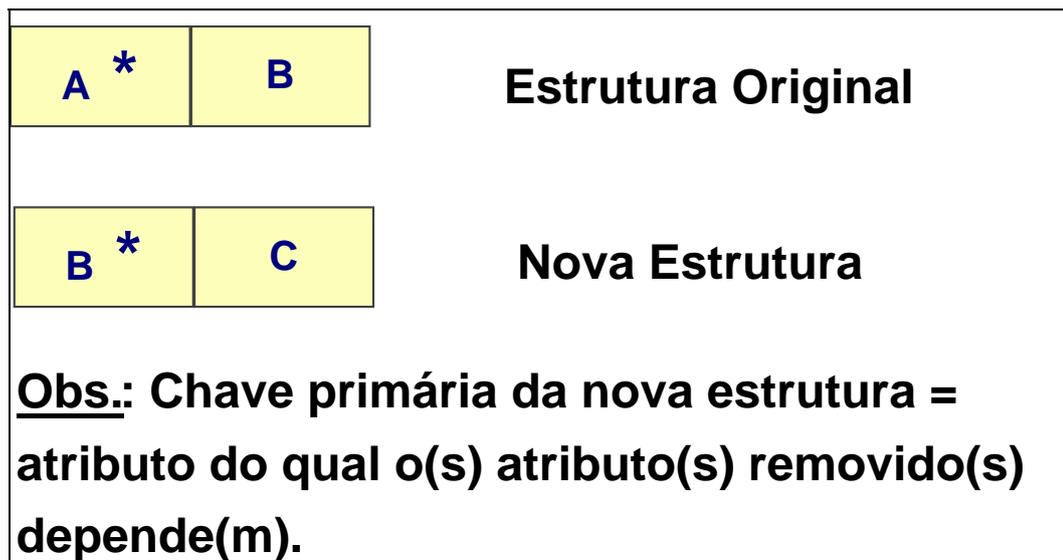


Fig. 3.9: Restruturação do modelo na terceira forma normal.

No exemplo apresentado, há uma dependência funcional do CPFCLIENTE com o NOMECLIENTE, ou vice-versa já que ambos são candidatos a definir unicamente um cliente. Dado o valor de NOMECLIENTE é possível determinar o valor de CPFCLIENTE portanto esse não depende da chave primária NUMPED. Para eliminar tal redundância, deve-se criar uma outra estrutura e colocar o modelo na terceira forma normal.

Pedido	Cliente
*NumPed DataPed ValorPed CodCliente	*CodCliente NomeCliente CPFCliente
ItemPedido	Produto
*NumPed *CodProduto Quantidade ValorItem	*CodProduto NomeProduto PreçoProduto

Fig. 3.10: Exemplo na terceira forma normal.

Existem outras formas normais, entretanto, colocando-se o modelo na terceira forma normal já é possível tratar os dados de forma consistente e segura provendo uma certa facilidade no desenvolvimento.

Propagação de chaves primárias

Outro processo que deve ser bem definido é quanto a forma de propagação de chaves entre as tabelas. No exemplo anterior pode-se observar que a chave primária da tabela Item foi formada pela propagação da chave primária das tabelas Pedido e Produto.

ItemPedido
*NumPed *CodProduto Quantidade ValorItem

Fig. 3.11: Tabela de Itens. Chave primária composta pela propagação das chaves primárias de outras tabelas.

Essa técnica de propagar as chaves primárias tornando-as também chaves primárias na tabela destino pode facilitar consultas diminuindo o número de tabelas utilizadas no comando. Imagine uma tabela de históricos de itens que armazenasse cada atualização que ocorresse em cada item. Seria fácil então consultar os históricos de um determinado pedido ou produto, sem a necessidade de utilizar uma outra tabela, já que essa tabela possui os campos NUMPED e CODPRODUTO.

Hist Itens
*NumPed
*CodProduto
*Data

Fig 3.12: Tabela de Hist. Itens

Entretanto, esse método pode tornar mais difícil a construção das aplicações de inclusão e manutenção dos dados.

Uma outra alternativa para montar a chave primária da tabela de *itens* é criar uma outra coluna com um número seqüencial do sistema e substituir a coluna CODPRODUTO. Desta forma o campo CODPRODUTO seria utilizado apenas para se fazer o relacionamento entre as tabelas.

ItemPedido
*NumPed
*NumSeq
CodProduto
Quantidade
ValorItem

Fig 3.13 Tabela de Itens com chave primária alternativa.

Assim, definindo novamente a tabela de histórico de itens ficaria da seguinte forma:

Hist Itens
*NumPed
*NumSeq
*Data

Fig 3.14 Tabela de Hist. De Itens.

Percebe-se, agora, que não há informação na tabela de histórico de itens sobre o produto. Para consultar os registros de um determinado produto é necessário fazer um “join” com a tabela de Itens, dificultando a consulta mas facilitando a construção das telas de manutenção.

Além disso, esta alternativa permite que o usuário altere o produto de um item já que esse não é mais parte da chave primária.

Concluindo, não existe uma melhor forma de se definir as chaves primárias. Existe a melhor alternativa para um determinado caso.

Ferramentas

Para modelar e definir a base de dados, várias ferramentas CASE fornecem um diagrama de “Entidade-Relacionamento” que permite definir as entidades que irão originar as tabelas do banco de dados e seus relacionamentos. Cria-se graficamente todos os elementos necessários, gerando uma documentação do modelo através de um dicionário de dados. Essas informações podem ser utilizadas para continuar o processo de definição do sistema. Em alguns casos, permitindo a geração da base de dados e até mesmo a geração da aplicação.

Criação da Base de Dados

Algumas ferramentas CASE permitem a geração do “script” da base de dados. Esse “script” é um arquivo texto com as definições das tabelas, colunas e outros elementos escritos na linguagem SQL suportada pela base de dados a ser utilizada. Através de ferramentas do sistema de banco de dados é possível criar a base de dados simplesmente abrindo esse arquivo “script” e executando-o.

Utilizando Interbase Windows ISQL

Para o Interbase da Borland pode-se utilizar a ferramenta Windows ISQL para lê o arquivo “script” e criar o banco de dados. O Windows ISQL é uma ferramenta que permite gerenciar, definir e manipular o banco de dados interativamente através de comandos SQL ou através de opções contidas no Menu.

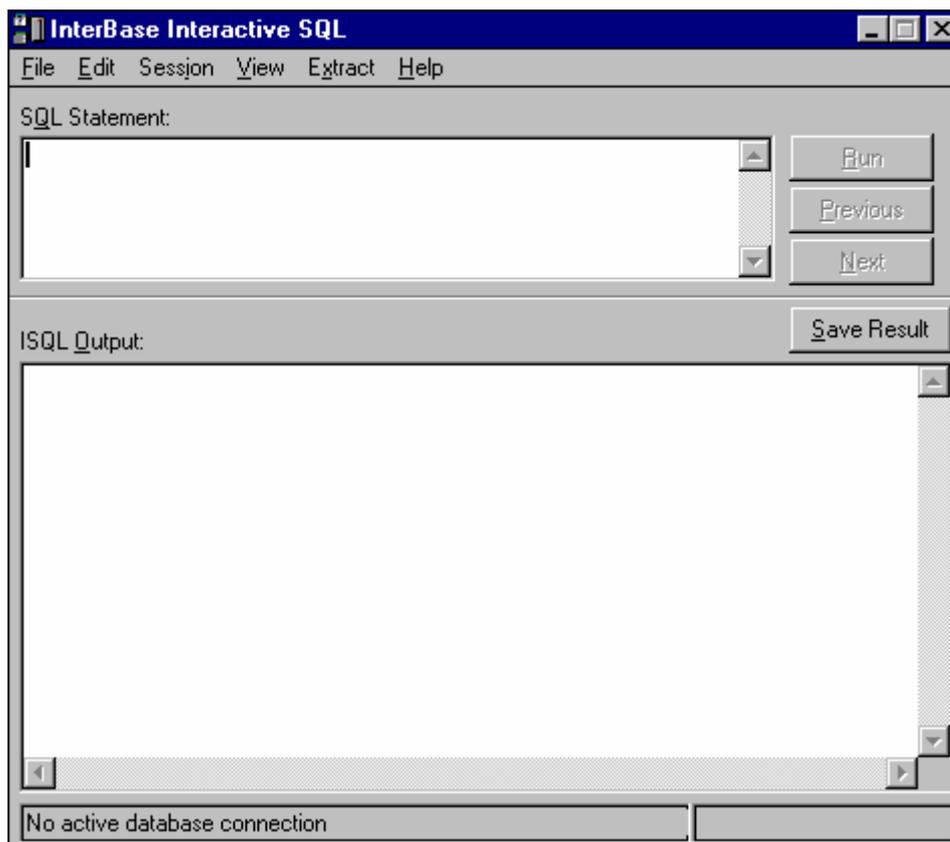


Fig. 3.15: Interactive SQL program.

Pode-se criar um novo banco de dados através da opção **File/Create Database** do Menu.

Use **Local Engine** para definir um banco de dados local e **Remote Server** caso o banco de dados esteja em um servidor remoto na rede. Na opção **Database** entre o nome do banco de dados com seu respectivo diretório. A seguir entre com um usuário/senha que tenha permissão para criar um novo banco de dados.

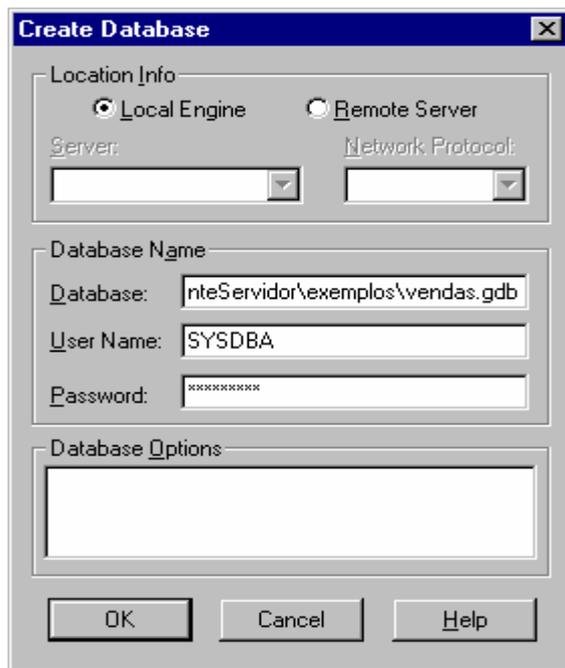


Fig. 3.16: Criação de um novo banco de dados.

A seguir deve-se executar o arquivo “script” vendas.sql através da opção do Menu **File/Run an ISQL Script**. Assim o programa irá criar as tabelas, colunas e outras definições presentes no arquivo. A estrutura criada é representada pelo modelo de dados da figura 3.12.

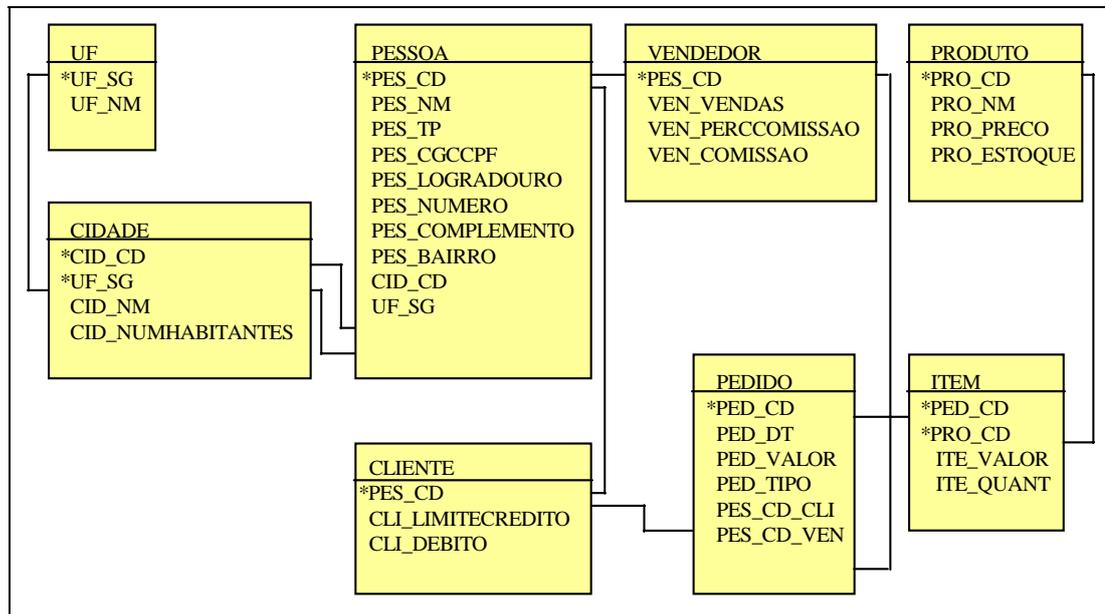


Fig. 3.17: Modelo de Dados Exemplo.

Linguagem SQL

Para acessar os bancos de dados relacionais foi desenvolvida uma linguagem chamada SQL. O Objetivo dessa linguagem é fornecer uma forma padrão de acesso aos bancos de dados, não importando a linguagem com que esses tenham sido desenvolvidos. Apesar da tentativa

de se tornar um padrão (ANSI), cada fornecedor hoje possui uma série de extensões que tornam as várias versões incompatíveis entre si. Por isso, não pense que uma aplicação construída para acessar um determinado banco de dados de um fornecedor, irá acessar também, sem qualquer modificação, o banco de dados de um outro fornecedor. Isto só é possível se a aplicação utilizar somente a parte comum (ANSI) da linguagem SQL, mas isto faz com que ela perca vários recursos importantes disponíveis em cada versão SQL de fornecedores diferentes. Alguns bancos de dados já suportam o padrão SQL ANSI-92 que já é mais abrangente numa tentativa de facilitar o processo de deixar transparente a base de dados utilizada pela aplicação. Entretanto, alguns fornecedores ainda não fornecem suporte ao SQL ANSI-92 de forma completa porque teriam que alterar partes estruturais de seus sistemas gerenciadores.

Categorias da Linguagem SQL

A linguagem SQL se divide em três categorias:

- DDL (Linguagem de Definição de Dados). Parte da linguagem com comandos para criação das estruturas de dados como as tabelas, colunas, etc. Ex: CREATE TABLE.
- DML (Linguagem de Manipulação de Dados). Parte da linguagem com comandos para acessar e alterar os dados armazenados no banco de dados. Os principais comandos dessa categoria são: SELECT, UPDATE, INSERT, DELETE.
- DCL (Linguagem de Controle de Dados). Parte da linguagem com comandos para definir usuários e controlar seus acessos aos dados. Ex: GRANT.

Utilizando o Windows ISQL para definir o Banco de Dados

O arquivo “script” utilizado para criação do banco de dados possui uma série de comandos SQL do tipo DDL. Esses comandos podem ser utilizados diretamente na linha de comando do ISQL. Um exemplo de SQL-DDL é o comando para criação de tabelas e colunas:

```
CREATE TABLE PEDIDO ( PED_CD CODIGO NOT NULL ,  
    PED_DT DATE NOT NULL ,  
    PED_VALOR VALOR ,  
    PED_TIPO TIPO NOT NULL ,  
    PES_CD_CLI CODIGO NOT NULL ,  
    PES_CD_VEN CODIGO NOT NULL ,  
    TIMESTAMP DATE ,  
PRIMARY KEY ( PED_CD ) ) ;
```

Esse comando cria a tabela pedido e suas colunas já identificando sua chave primária através da cláusula **primary key**.

Utilizando o Windows ISQL para acessar o Banco de Dados

Para se conectar ao banco de dados, pode-se utilizar a opção **File/Connect to Database**. Deve-se selecionar o banco de dados e entrar com o usuário e senha de acesso.



Fig. 3.18: Conexão com o Banco de Dados.

Para executar um comando SQL, basta escrevê-lo na seção **SQL Statement**. O comando seguinte insere um registro na tabela de produto. Para executar o comando, pode-se utilizar o botão **Run**.

```
insert into PRODUTO (PRO_CD, PRO_NM, PRO_ESTOQUE, PRO_PRECO) values
(1, 'Impressora', 5, 200.00)
```

Para verificar a inserção, pode-se selecionar o registro através do comando seguinte e pressionar novamente o botão **Run**:

```
select * from PRODUTO
```

ou

```
select PRO_CD, PRO_NM, PRO_ESTOQUE, PRO_PRECO from PRODUTO
```

Para alterar os dados de um registro existente, pode-se utilizar o comando **Update**:

```
update PRODUTO set PRO_PRECO = 250.00, PRO_ESTOQUE = 6 where PRO_CD= 1
```

Para verificar os dados novamente, pode-se pressionar o botão **Previous** até encontrá-lo e a seguir pressionar o botão **Run**.

Finalmente, para excluir um registro, pode-se utilizar o comando **Delete**:

```
delete from PRODUTO where PRO_CD = 1
```

Para concluir as operações feitas e registrá-las no banco, de modo que outros usuários sejam capazes de vê-las, utiliza-se o comando:

`commit`

Consistência e Integridade dos Dados

A maioria dos bancos de dados relacionais possuem recursos para consistir e tornar íntegro os dados armazenados em suas tabelas. Desta forma, qualquer ferramenta que o usuário utilizar para acessá-lo é obrigada a respeitar as regras, mantendo a integridade dos dados. Os recursos de consistência variam de banco para banco, mas pode-se definir conceitualmente algumas categorias de integridade como: integridade referencial, domínios e regras do negócio.

Integridade Referencial

Integridade referencial é um conjunto de regras e de consistências entre os registros de duas tabelas que se relacionam. Como foi visto no modelo relacional, quando duas tabelas se relacionam, a chave primária de uma é copiada para a outra e se esses dados forem alterados ou excluídos da tabela original é necessário verificar o que será feito com os dados e registros duplicados na outra tabela. Quando se define uma integridade referencial, está se definindo o procedimento que será tomado quando esses processos ocorrerem.

Sejam duas tabelas “A” e “B” que se relacionam através de uma coluna “c” que é a chave primária de “A” e portanto foi repetida em “B” para se fazer o relacionamento. Quando se define uma integridade referencial para esse relacionamento, está se definindo que a coluna “c” da tabela “B” só pode conter valores já cadastrados na coluna “c” da tabela “A”.

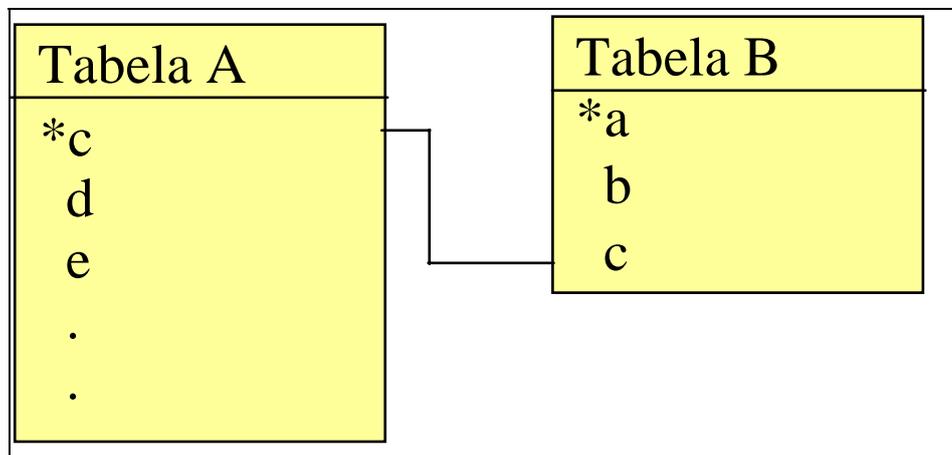


Fig 3.19: Integridade Referencial.

Existem três formas de se manter essa regra quando registros da tabela “A” são excluídos:

- **Restrict:** Define que se o valor da coluna “c” de “A” existir em algum registro de “B”, o registro não poderá ser excluído e uma mensagem de erro retornará para a aplicação;
- **Cascade:** Define que se o valor da coluna “c” de “A” existir em algum registro de “B”, todos os registros que possuírem esse valor serão também excluídos;

- **Set Null:** Define que se o valor da coluna “c” de “A” existir em algum registro de “B”, os valores de “c” em todos os registros serão transformados para “Null”;

Em todos os casos, se o usuário tentar inserir um registro em “B” com um valor de “c” que não exista na tabela “A”, um erro será reportado à aplicação.

Alguns bancos, também permitem fazer o mesmo tratamento quando a chave da tabela for alterada, ao invés de excluída. Entretanto, essa alteração de chaves não deve ser um processo comum nas aplicações, sendo normalmente proibidas.

Os bancos de dados possuem formas diferentes de disponibilizar esses serviços. Alguns possuem até mais de uma forma de fazer o mesmo processo. Mas de maneira geral existem duas possibilidades de implementação:

Integridade Referencial Declarativa

Essa é uma forma mais fácil, porém menos flexível. Com apenas um comando se define a integridade. Porém, alguns bancos não suportam essa sintaxe ou a fornecem de maneira limitada somente para regras do tipo “Restrict”. A maior vantagem dessa alternativa é sua simplicidade e por isso menos sujeita a erros de implementação.

Triggers

Os triggers são pedaços de código escritos em uma extensão da linguagem SQL fornecida por cada fornecedor, sendo portanto uma linguagem proprietária. Essa extensão possui instruções para implementar “loops” e verificar condições permitindo fazer pequenos programas estruturados. Os triggers são disparados automaticamente pelo banco quando eventos de inclusão, alteração ou exclusão ocorrem em uma determinada tabela. Portanto, através dos triggers pode-se desenvolver a lógica necessária para se manter a integridade referencial entre as tabelas.

Domínio dos dados

Outro tipo de consistência é com relação ao domínio dos dados. Para um determinado campo de dado pode existir um conjunto de valores permitidos e o banco de dados deve reportar um erro à aplicação se um valor inválido for informado. Um tipo de consistência de domínio que é normalmente fornecido pelos bancos é a possibilidade do campo ser nulo ou não. Alguns bancos possuem também comandos para definir um conjunto de valores válidos para um determinado campo. Ex: Valores (‘J’,‘F’) para um campo que armazena se a pessoa é física ou jurídica. Outros possuem uma sintaxe para definir valores mínimos e máximos. Pode-se, ainda, utilizar o recurso de triggers para esse propósito.

Regras de Negócio

Pode-se definir como regras de negócio a integridade que deve existir entre os valores de mais de um campo de uma ou mais tabelas. Pode-se considerar como regra do negócio as consistências e atualizações que devem ser feitas em várias tabelas, quando um registro é inserido, alterado ou excluído de uma tabela. Ou um processo de cálculo disparado sobre os dados do banco que também fazem atualizações em diferentes locais, mas de forma a manter os valores dos dados sempre íntegros e consistentes. Um exemplo seria a atualização de estoque que normalmente envolve outras tarefas em um sistema de controle de estoque.

Para implementar essas regras de negócio no banco de dados são utilizados os recursos de triggers e stored procedures. Stored procedures são blocos de código assim como os triggers.

A diferença entre eles é que os triggers são disparados automaticamente por inserções, alterações e exclusões feitas nas tabelas, enquanto as stored procedure são chamadas explicitamente pela aplicação.

Utilizando o Windows ISQL para definir integridades e consistências

O arquivo “script” utilizado na criação do banco de dados exemplo também possui comandos que definem regras de integridade e domínios que serão seguidas pela base de dados:

```
CREATE TABLE PEDIDO ( PED_CD CODIGO NOT NULL,  
    PED_DT DATE NOT NULL,  
    PED_VALOR VALOR,  
    PED_TIPO TIPO NOT NULL,  
    PES_CD_CLI CODIGO NOT NULL,  
    PES_CD_VEN CODIGO NOT NULL,  
    TIMESTAMP DATE,  
PRIMARY KEY ( PED_CD ) );
```

Na própria definição da tabela já é atribuída a verificação de nulidade para alguns campos através da sintaxe **Not Null**.

O próximo comando define um conjunto de valores possíveis para o campo PES_TP da tabela PESSOA.

```
ALTER TABLE PESSOA ADD CHECK( ( PES_TP='J' ) OR ( PES_TP='F' ) )
```

Para definir a integridade referencial entre duas tabelas foi usada a forma declarativa através do comando:

```
ALTER TABLE ITEM ADD FOREIGN KEY ( PED_CD ) REFERENCES PEDIDO(PED_CD) ;
```

Esse comando define que um item só pode ser incluído se o valor da coluna PED_CD da tabela de ITEM já existir na tabela PEDIDO. Além disso, só é possível excluir um PEDIDO que não possua ITENS associados. A sintaxe do Interbase para definição de integridade referencial declarativa somente permite a forma “restrict”. As demais formas teriam que ser definidas através de triggers.

Utilizando o Windows ISQL para testar as consistências.

O modelo exemplo de pedido possui definições para fazer a integridade referencial entre suas tabelas. Uma delas é na relação que existe entre as tabelas UF e CIDADE como mostrada na figura 3.13. Esta integridade não permite então inserir um registro na tabela CIDADE se o valor atribuído a UF_SG não existir na tabela UF. Além disso, se tentarmos excluir um registro da tabela UF que esteja sendo referenciado pela tabela CIDADE irá retornar uma mensagem de erro, já que o tipo da integridade é “restrict”.

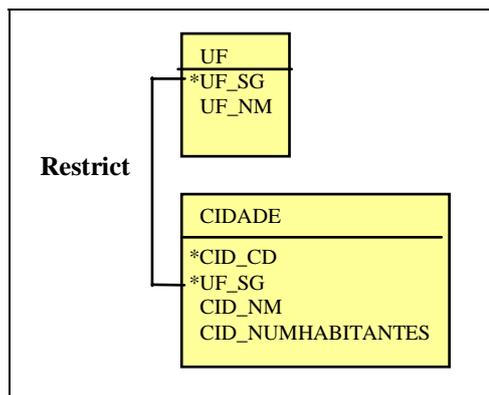


Fig.3.20: Integridade Referencial

Executando o seguinte comando no Windows ISQL...

```
insert into CIDADE(CID_CD,UF_SG,CID_NM,CID_NUMHABITANTES) values
(1,"MG","Belo Horizonte", 500000)
```

uma mensagem de erro irá retornar já que não existe um registro com UF_SG = 'MG' na tabela UF, portanto deve-se primeiro inserir o registro:

```
insert into UF(UF_SG,UF_NM) values ("MG", "Minas Gerais")
```

A seguir pode-se inserir novamente o registro na tabela CIDADE

```
insert into CIDADE(CID_CD,UF_SG,CID_NM,CID_NUMHABITANTES) values
(1,"MG","Belo Horizonte", 500000)
```

Assim, os dois registros foram inseridos com sucesso, e deve-se portanto efetivar as inserções utilizando o comando **commit**. Pode-se tentar excluir o registro da tabela UF através do comando:

```
delete from UF where UF_SG = "MG"
```

Pelo mesmo motivo visto anteriormente, o registro também não pode ser excluído. Deve-se então excluir primeiro o registro de CIDADE para depois poder excluir o de UF.

O modelo exemplo também possui algumas consistências de domínio, principalmente em relação a nulidade. Pode-se tentar inserir na tabela UF utilizando o seguinte comando:

```
insert into UF(UF_SG,UF_NM) values ( Null, Null)
```

Esse comando deve retornar erro informando que UF_SG não pode ser nulo. Se o comando for alterado para:

```
insert into UF(UF_SG,UF_NM) values ( "SP", Null)
```

irá continuar apresentando erro, agora na coluna UF_NM que também não pode conter nula.

Distribuição da Consistência e Integridade dos Dados

Como foi visto, é possível implementar várias regras de consistência dos dados no próprio banco de dados. Entretanto, é possível implementá-las também na aplicação e portanto fica uma dúvida de qual seria o melhor lugar para implementá-las. Ambas as alternativas possuem vantagens e desvantagens que devem ser observadas no início do processo de desenvolvimento.

A principal vantagem em se colocar todas as regras no banco de dados é que esse mantém a consistência e integridade dos dados independente da ferramenta de acesso utilizada pelo usuário. Além disso, as manutenções dessas regras ficam localizadas em um único local, ao invés de ficarem espalhadas por toda a aplicação ou por diversas aplicações. Entretanto, o desenvolvimento utilizando essa linha de trabalho é muito mais árduo e demorado. As ferramentas disponíveis nesse ambiente não apresentam ainda muitos recursos que dêem produtividade na construção das regras. As linguagens utilizadas para escrever as “stored procedures” e “triggers” são proprietárias dos fornecedores, tornando a empresa dependente de único fornecedor, dificultando a distribuição dos dados em bancos de fornecedores diferentes. Além disso é necessário dividir o desenvolvimento em duas linguagens. A linguagem da ferramenta para se montar a interface e a chamada dos processos e a linguagem para construção das “stored procedures”.

Atualmente, o mercado se encontra dividido com cada empresa tomando um caminho diferente para distribuir essas regras entre o cliente e o servidor. Mas não pode-se dizer que um caminho já seguido por uma empresa que obteve sucesso é o mais indicado para uma outra empresa. Muitas outras variáveis precisam ser consideradas e dificilmente serão iguais entre duas empresas. Além disso, as ferramentas e a tecnologia evoluem fazendo como que um mesmo caminho possa obter resultados diferentes, considerando-se a tecnologia atualmente disponível.

De uma maneira geral, as duas primeiras categorias de consistência (integridade referencial e domínios) são normalmente implementadas no banco de dados, sendo que os domínios são implementados repetidamente na aplicação para evitar que os comandos sejam enviados ao banco. A grande dúvida mesmo, é com relação às regras de negócio que algumas empresas implementam totalmente no banco de dados, outras totalmente na aplicação e algumas distribuem entre esses dois componentes. Atualmente, existe mais um alternativa na nova geração da arquitetura Cliente/Servidor: a implementação das regras de negócio em um novo componente chamado “Servidor de Aplicações”.

SQL Explorer

Esse capítulo apresenta a ferramenta de definição e manipulação de dados SQL Explorer da Borland.

Pode-se utilizar a ferramenta SQL Explorer para criar a estrutura do banco de dados de maneira gráfica e interativa. Além disso, essa ferramenta permite definir novos aliases para os bases de dados, manipular os dados (inserir, alterar, excluir), e construir um dicionário de dados com definições que podem ser utilizadas para facilitar a construção de aplicações em Delphi.

Criação de Alias

Uma das possibilidades do SQL Explorer é a definição de “aliases”. Ao abrir o SQL Explorer é exibido na lista **Databases** todos os “aliases” definidos no BDE.

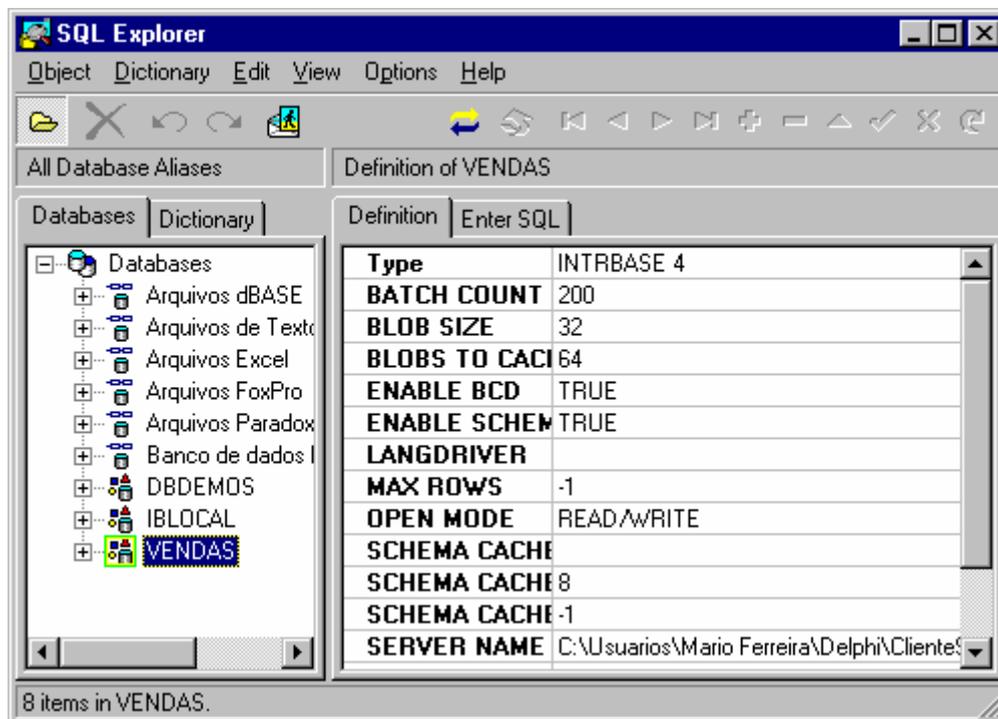


Fig 4.1: Lista de aliases no SQL Explorer

Para criar um novo alias, deve-se pressionar o botão direito do “mouse” em qualquer região do SQL Explorer para exibir um “popup menu” e selecionar a opção **New...** Uma tela será exibida para que seja informado o tipo de “driver” do novo alias.

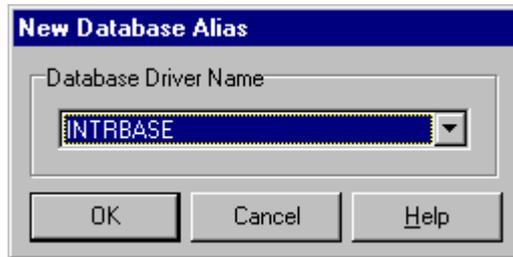


Fig. 4.2: Definição de um novo alias.

Visualização e Edição de Dados

Para visualizar e editar os dados de um banco de dados, deve-se expandir sua lista de opções exibindo seus componentes. Cada tipo de “driver” de banco de dados pode possuir opções diferentes na lista, entretanto uma opção comum a todos é: **Tables**. Ao expandir também essa opção, será listada todas as tabelas contidas nesse banco de dados. Pode-se, então, selecionar uma das tabelas e exibir seus dados através da ficha **Data** do lado direito da tela. Na barra de ferramentas existem botões que permitem a navegação entre os registros, inclusão, edição e exclusão dos dados.

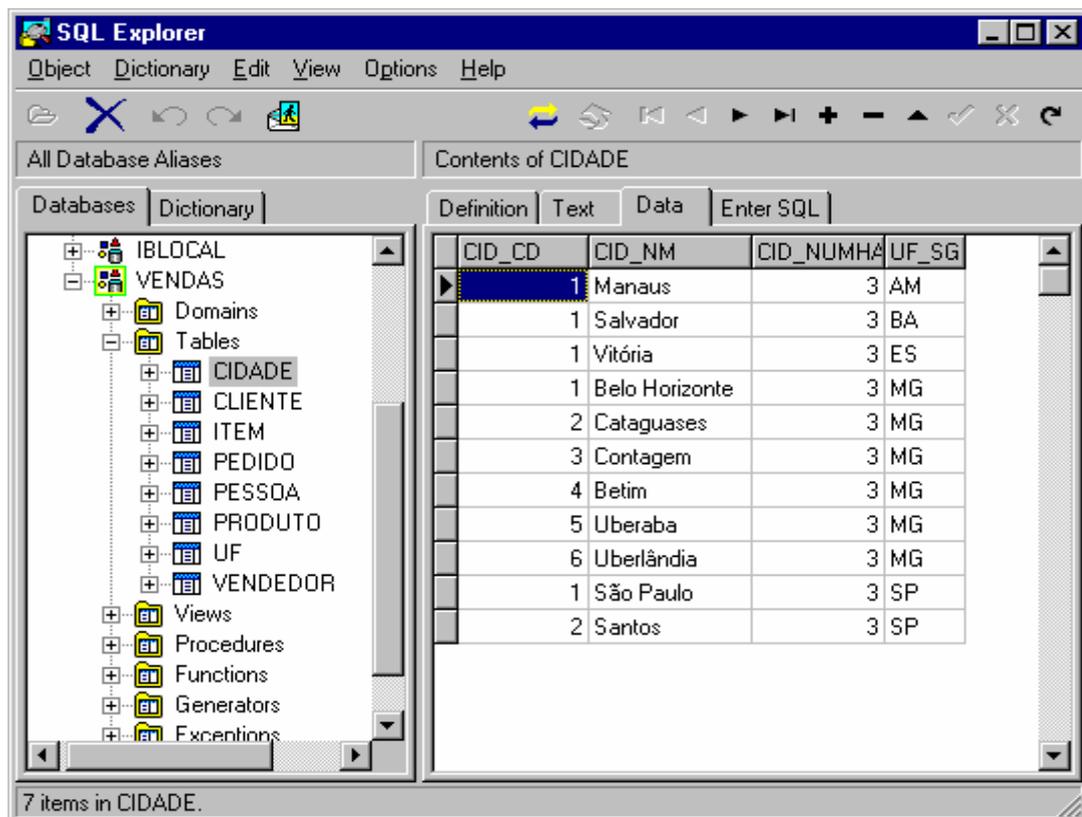


Fig 4.3: Visualização e edição dos dados.

Pode-se também utilizar comandos SQL par visualizar e editar os dados através da ficha **Enter SQL**.

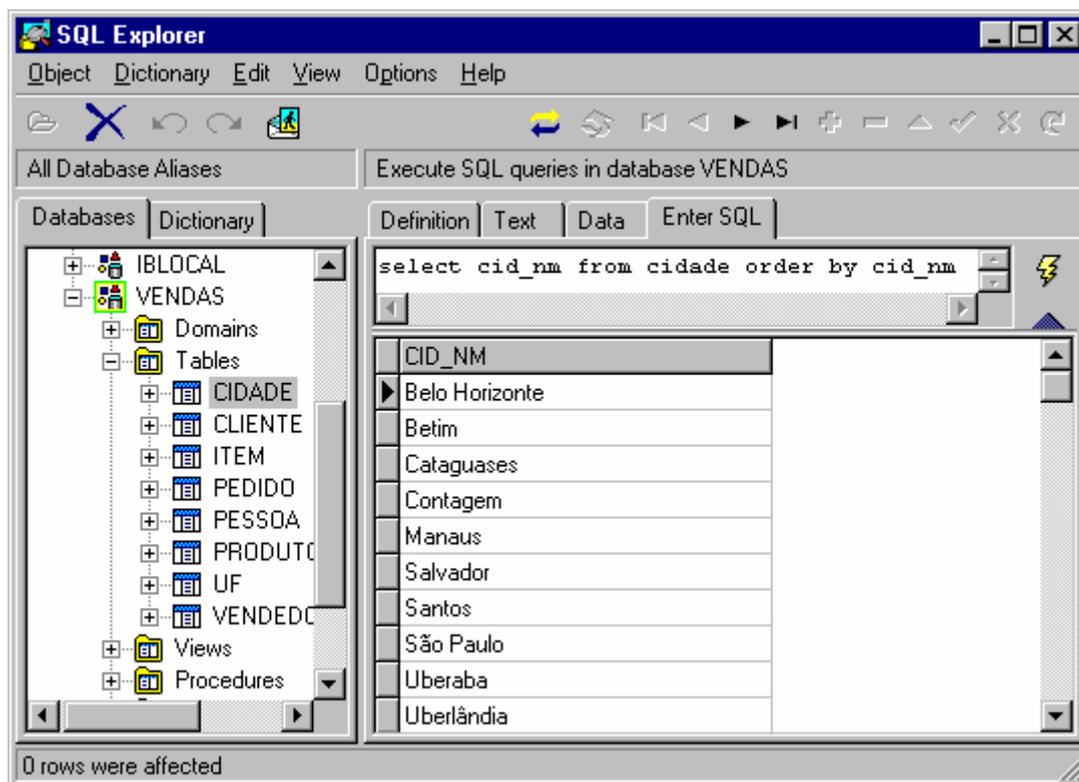


Fig 4.4: Visualização de dados através de SQL.

Pode-se também manipular os dados através de comandos SQLs.

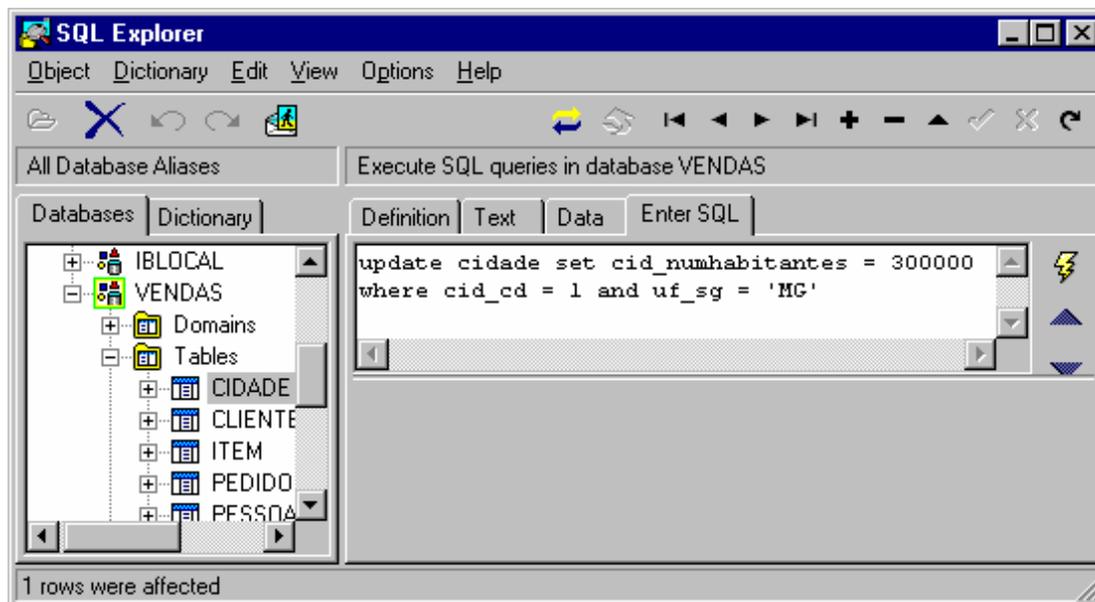


Fig. 4.5 Edição de dados através de SQL.

Definição de novos elementos

Através do SQL Explorer pode-se também definir novos elementos na estrutura do banco de dados. Para criar uma nova tabela no banco de dados, deve-se pressionar o botão direito sobre o item **Tables** e selecionar a opção **New....** no “popup menu”. Uma nova tabela é criada com um nome default. Pode-se então alterar seu nome e definir os demais elementos que a compõem.

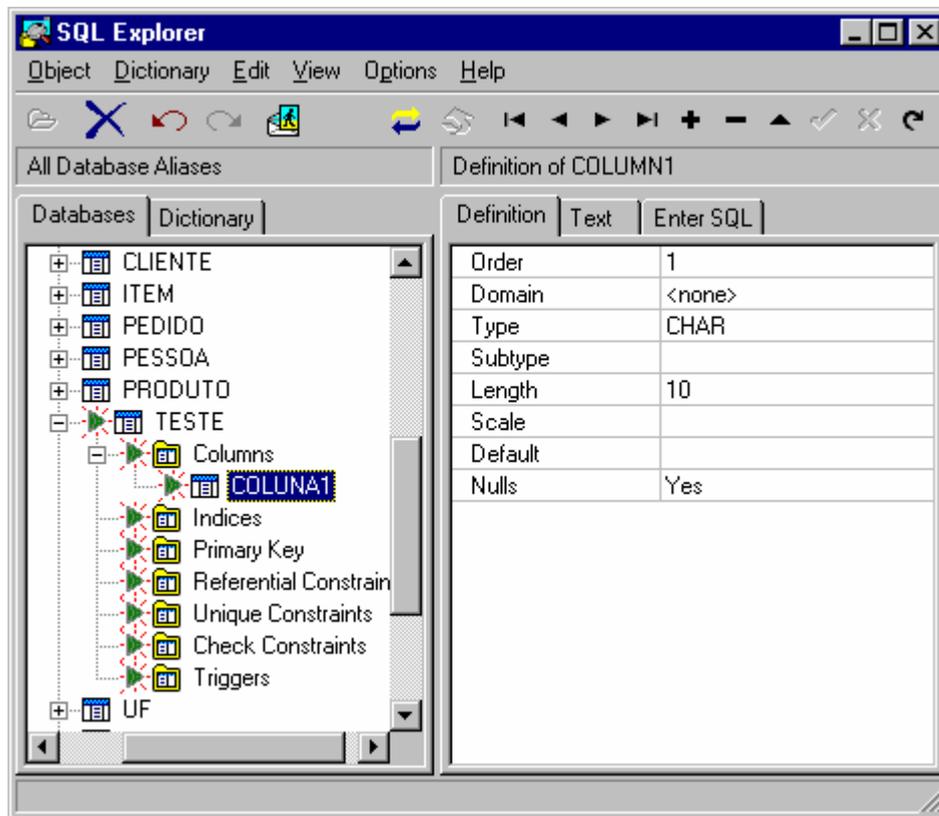


Fig 4.6: Definição de uma nova tabela.

Pode-se definir as colunas, chave primária, integridade referencial, “triggers”, índices e outros elementos para as tabelas através do SQL Explorer para os vários tipos de banco de dados.

Definição de Dicionários de dados

Através do SQL Explorer pode-se definir um dicionário de dados. Esse dicionário é um banco de dados especial usado para armazenar um conjunto de definições dos itens de dado existente nas tabelas do banco de dados da aplicação. Esse conjunto de definições descreve as propriedades do componente *Field* de um *DataSet* no Delphi, o tipo do campo e o tipo de controle visual a ser criado quando o componente *Field* é arrastado para a tela. Desta forma, consegue-se definir uma única vez as propriedades de um domínio de dado que pode estar sendo utilizado por diversos itens em vários locais da aplicação.

Criação de um novo Dicionário

Para criar um novo dicionário de dados, deve-se selecionar a ficha **Dictionary** do lado esquerdo da tela, pressionar o botão direito e escolher a opção **New** do Menu. A seguir será

exibida uma tela para se definir o nome do dicionário, o alias do banco aonde a tabela do dicionário será criada e o nome da tabela.

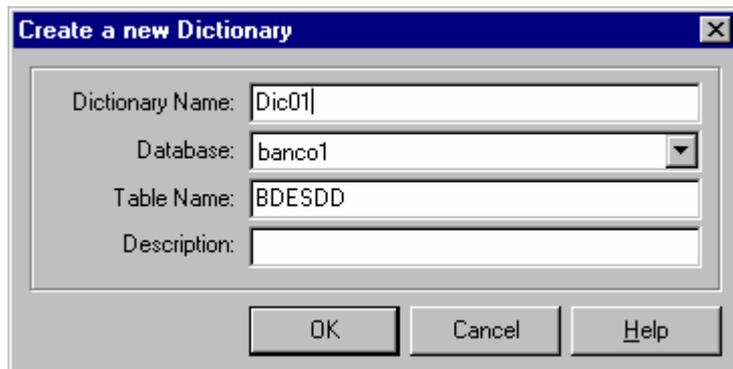


Fig 4.7: Criação do Dicionário de Dados

Importação das definições do Banco de Dados

Para importar os atributos e definições do banco de dados da aplicação deve-se selecionar a opção **Dictionary/Import From Database**.

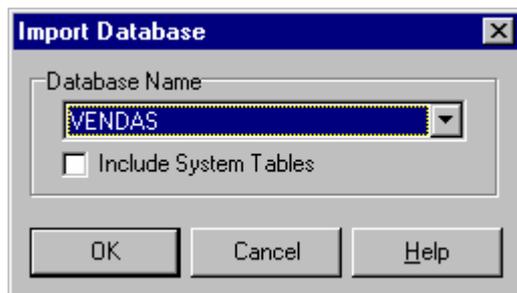


Fig 4.8: Importação das definições do Banco de Dados.

Definição das propriedades dos *Attribute Sets*

A seguir são exibidas as definições importadas e os “*Attribute Sets*” encontrados pelos “constraints” definidos no banco de dados para cada elemento.

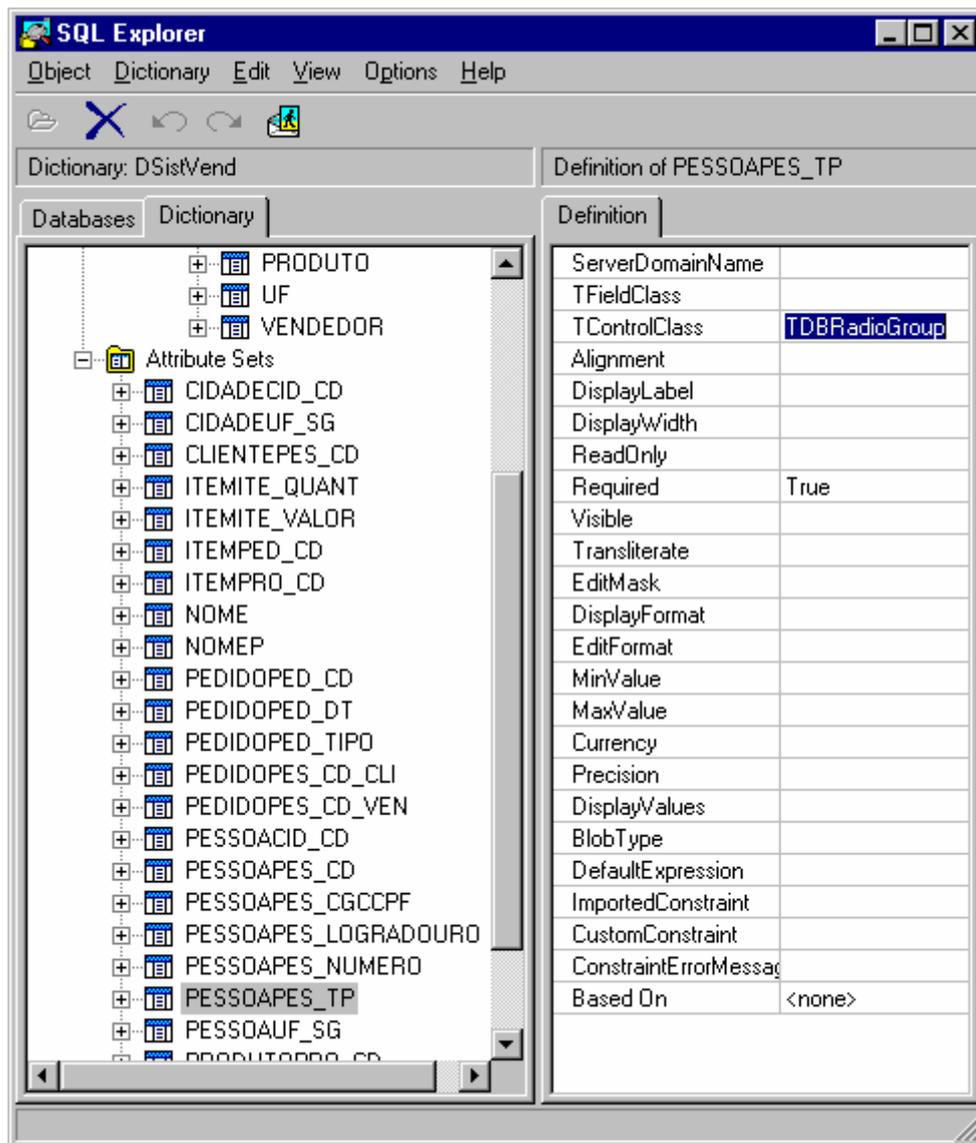


Fig 4.8: Definição de propriedades dos *Attribute Sets*.

Utilização do Dicionário no Delphi

Para utilizar as definições estabelecidas no dicionário em uma aplicação Delphi, pode-se arrastar os campos desejáveis do SQL Explorer para a tela da aplicação. Automaticamente, serão criados um *DataSet* e um *Datasource* para cada tabela do banco de dados.

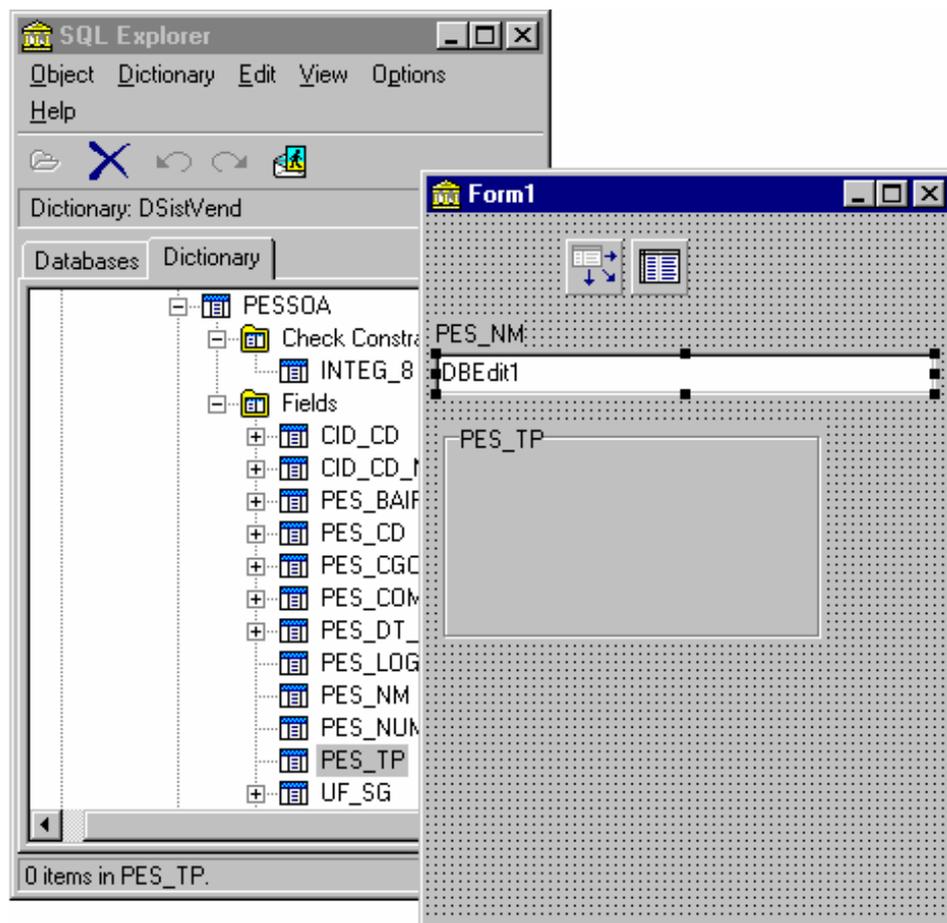


Fig 4.9: Criação da tela através do Dicionário do SQL Explorer.

Além disso, pode-se também associar um componente *Field* através da opção **Associate attributes...** do “popup menu”.

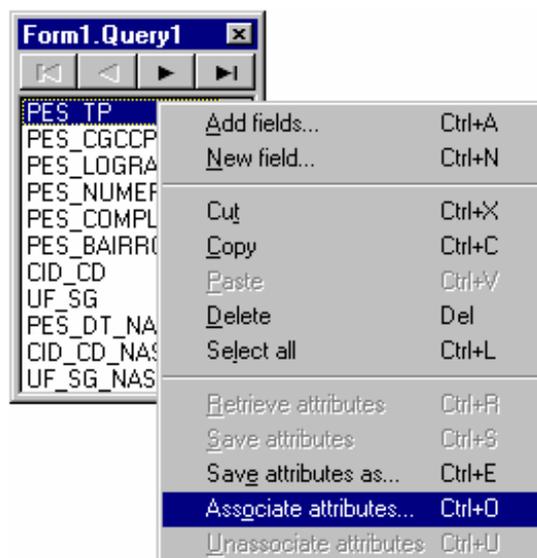


Fig 4.10: Associação de *Fields* com o dicionário.

Trabalhando com Bancos de Dados Relacionais

Esse capítulo apresenta a forma de trabalho dos bancos de dados relacionais.

Os bancos de dados relacionais fornecem um conjunto de serviços para que as aplicações possam acessá-los e manipulá-los. Portanto, é importante entender bem sua funcionalidade para melhor aproveitar seus recursos nas aplicações. Existem algumas diferenças na forma de trabalhar entre os fornecedores, mas grande parte dos conceitos podem ser aplicados a todos os bancos de dados relacionais.

Componentes da Arquitetura Cliente/Servidor-dlsf

Como foi visto, a arquitetura Cliente/Servidor é composta de dois componentes físicos que se comunicam através da rede: a estação de trabalho do usuário e o servidor de banco de dados.

Para se estabelecer a comunicação entre esses dois componentes são utilizadas várias camadas de software que são instaladas em cada componente físico. A estação de trabalho cliente deve ter, além da aplicação final, vários outros elementos para acessar a base de dados em um SGBD através de rede.

- **Database Engine:** biblioteca fornecida pelo fornecedor da ferramenta de desenvolvimento com o objetivo de fornecer uma forma única e transparente da aplicação acessar diferentes bases de dados. Ex: BDE (Borland Database Engine);
- **SQL Links:** “driver” fornecido também pelo fornecedor da ferramenta de desenvolvimento responsável pela comunicação do Database Enginer com uma base de dados específica. Sua principal característica é traduzir os comandos utilizados pelo Database Enginer para comandos conhecidos pela base de dados utilizada. Ex: SQL links para acessar Oracle, Sybase, Informix, MS SQL Server, Interbase, etc. A aplicação pode optar por utilizar o padrão ODBC para acessar a base de dados, ao invés de utilizar SQL Links (acesso nativo). Entretanto, o acesso feito pelos SQL Links ainda possui um desempenho superior em relação ao acesso feito via “driver” ODBC.

- **Client do Banco de Dados:** API fornecida pelo fornecedor do SGDB e instalada na estação cliente para estabelecer a comunicação com o banco de dados. Nessa API se encontram as funções de acesso a base de dados. É também, responsável por utilizar um determinado protocolo de rede para encontrar o servidor de banco para que a aplicação possa acessá-lo enviando comandos e buscando os dados.
- **Protocolo de Rede:** softwares responsáveis pela transmissão dos dados pela rede entre a máquina cliente e o servidor.

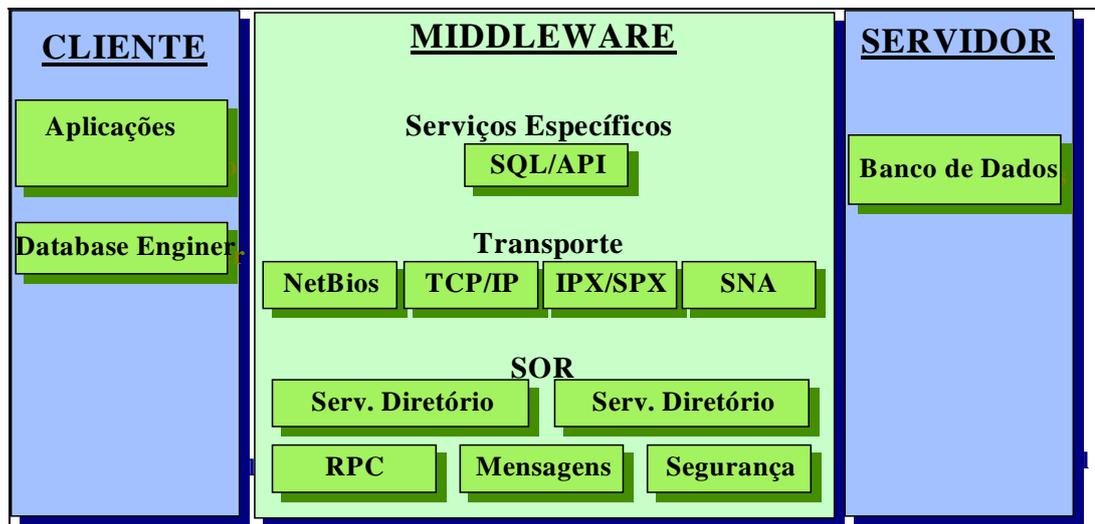


Fig 5.1: Componentes da comunicação entre o Cliente e o Servidor.

Conexões e Contextos

Pode-se definir como conexão um caminho estabelecido entre a aplicação e o banco de dados através de um usuário e sua senha. Para cada conexão é criada uma estrutura na memória do servidor de banco preparada para receber os comandos SQL enviados pela aplicação.

Cada conexão pode ter um ou mais contextos. Para executar um comando SQL na aplicação é necessária a criação de pelo menos um contexto dentro da conexão.

Os comandos SQLs enviados por um contexto passam por pelo menos duas fases: preparação e execução. Na primeira fase é verificada a sintaxe do comando e é definido o caminho de acesso que será utilizado pela execução do comando (otimização). Na segunda fase o comando é finalmente executado. Depois que a primeira fase for completada, pode-se repetir várias vezes a execução da segunda fase, otimizando o processo, já que o tempo gasto para compilar e otimizar o comando não será gasto novamente. Entretanto, um mesmo contexto pode ser usado para a execução de mais de um comando SQL. Quando um outro comando SQL é enviado pelo mesmo contexto, o comando anterior deixa de existir e uma nova necessidade de execução desse comando terá que passar novamente pelas duas fases. Portanto um contexto só permite controlar e manter preparado um único comando SQL. Comandos anteriores não são mais vistos pelo contexto.

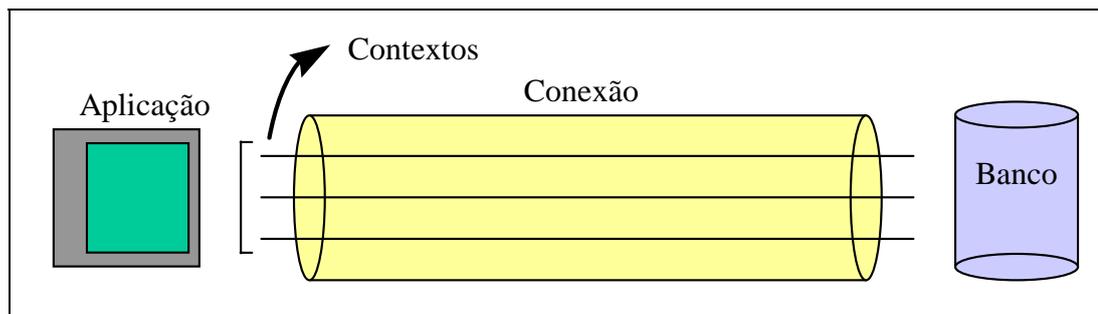


Fig 5.2: Conexões e Contextos

Para resolver esse problema, alguns bancos permitem a utilização de múltiplos contextos dentro de uma mesma conexão, possibilitando o controle de mais de um comando SQL ao mesmo tempo. Para os bancos que não possuem múltiplos contextos, é necessário criar várias conexões para se ter o mesmo recurso. Porém existem algumas diferenças entre essas duas formas de utilização. Uma conexão é totalmente independente de uma outra conexão, como se fossem dois usuários diferentes acessando o banco de dados. Já vários contextos dentro de uma conexão são interligados e cooperam para a execução de um mesmo processo.

Conexões e Contextos no Delphi

Para acessar a base de dados, deve-se primeiramente criar um “alias” no BDE para o banco de dados exemplo no interbase. Pode-se atribuir o nome “vendas” para o “alias”, definir o parâmetro **server name** para o nome do banco de dados e seu diretório, **user name** para SYSDBA.

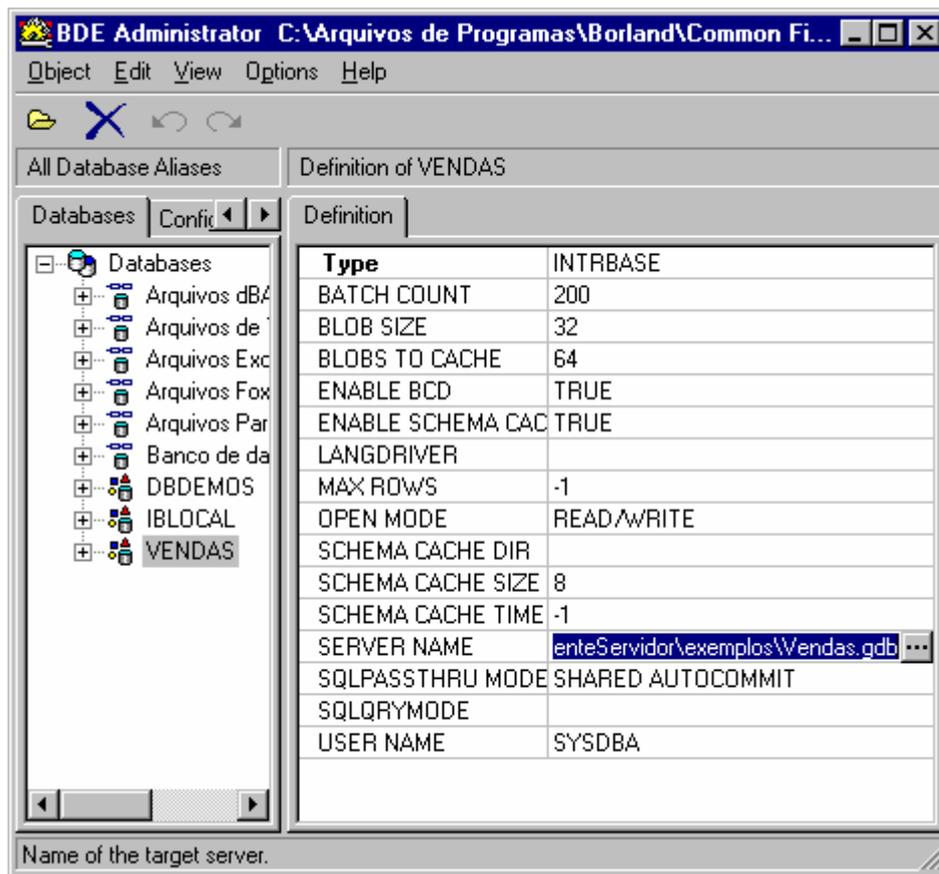


Fig 5.3: Configuração do alias no BDE.

TDatabase e TDataSet

O componente do Delphi que representa uma conexão com o banco de dados é o *TDatabase*. Deve-se utilizar um componente *TDatabase* para cada conexão que se deseje fazer com o banco. Alguns bancos controlam o número de licenças de usuários através do número de conexões. Portanto, o número de conexões deve ser bem reduzido.

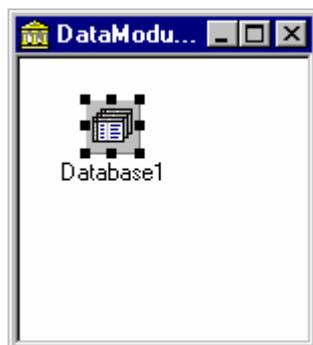


Fig 5.4: Componente TDatabase do Delphi.

Os componentes *TQuery* e *TTable* são utilizados para fornecer contextos dentro de uma conexão do banco de dados. Através da propriedade **DatabaseName** é possível ligá-los a um componente *TDatabase*.

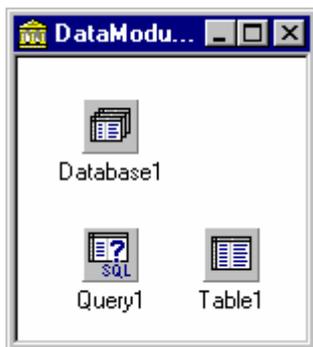


Fig 5.5: Os componentes *TQuery* e *TTable* do Delphi.

Cursors e Result Sets

Quando o SQL executado é um comando **select** e retorna mais de um registro, é necessário que a aplicação abra um cursor para poder percorrer e trazer todos os registros do banco de dados para a aplicação. Desta forma é criado um “result set”, que é uma estrutura temporária com a indicação dos registros que foram selecionados. O cursor é inicialmente posicionado na primeira linha e a aplicação pode requisitar que o cursor avance linha a linha no “result set” até que seu final seja alcançado. Os cursores e “result sets” podem ser criados no banco de dados se estes suportarem ou na própria estação de trabalho. A principal utilidade dos cursores e “result sets” é permitir percorrer os dados em qualquer sentido e poder fazer atualizações de registro através do posicionamento do cursor.

Se nenhum cursor for declarado explicitamente no banco de dados, o banco de dados cria um cursor internamente apenas para enviar seqüencialmente os dados para a aplicação quando estes forem requisitados. Assim, não é necessário que o servidor envie todos os dados selecionados de uma só vez para aplicação. Cabe a aplicação dizer quando cada registro de dados deve ser trazidos para a máquina cliente.

Normalmente, é utilizado “buffers” que pode conter mais de um registro para enviar os dados para aplicação. O tamanho desses “buffers” pode ser redefinido para otimizar os pacotes de dados enviados pela rede.

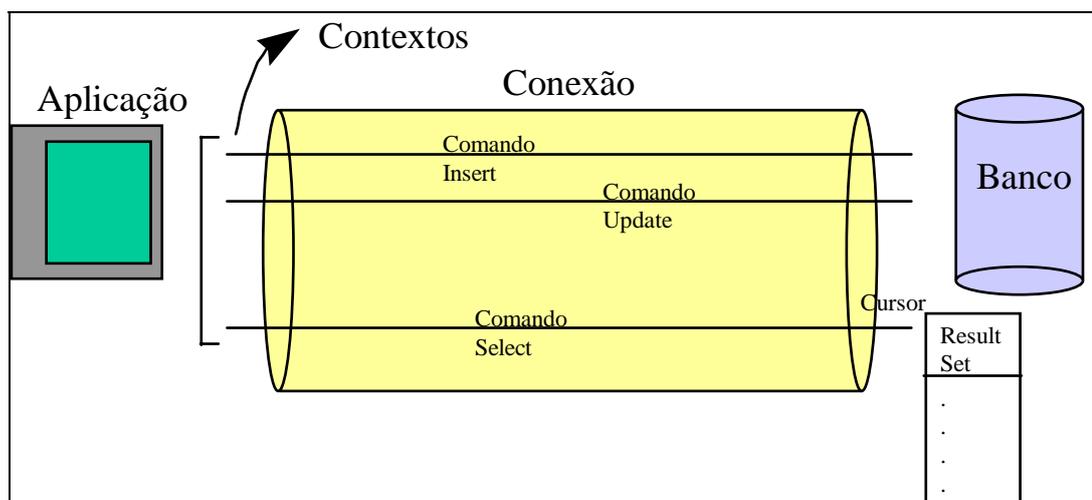


Fig 5.6: Cursores e Result Sets

O comando **select**, como qualquer outro comando é executado em um determinado contexto de uma conexão. Então, se outro comando SQL utilizar o mesmo contexto antes que todas as linhas do comando **select** anterior tenham sido trazidas para aplicação o cursor e seu “result set” serão destruídos e o restante das linhas é perdido. Portanto, para se manter o “result set” de um comando **select** até que todas as linhas sejam trazidas, é preciso abrir um contexto de maneira exclusiva para esse comando.

A medida que as linhas são trazidas para aplicação, normalmente são criados caches na memória ou em disco para armazenar os dados permitindo a navegação entre registros (inclusive registros anteriores). Esse recurso é normalmente conhecido como “Front End Result Set” por estar localizado na máquina cliente.

Cursor e Result Sets no Delphi

Os componentes *TTable* e *TQuery* do Delphi utilizam esses recursos para acessar o banco de dados. Quando se abre uma “Query” ou uma “Table” no Delphi, um cursor apontando para um “result set” é criado e as linhas são trazidas e armazenadas na aplicação a medida que o usuário percorre os registros até o final da consulta. Portanto, se o usuário selecionar um número grande de registros (1.000.000), ele irá esperar apenas o tempo necessário para compilar e executar a consulta e trazer alguns poucos registros suficientes para popular a parte visível dos dados na tela. Os demais registros serão trazidos a medida que o usuário navegue em direção ao final da consulta. Se o usuário executar um comando para ir para o final da consulta, poderá ter que esperar o tempo necessário para se trazer todos os 1.000.000 de registros para a estação cliente.

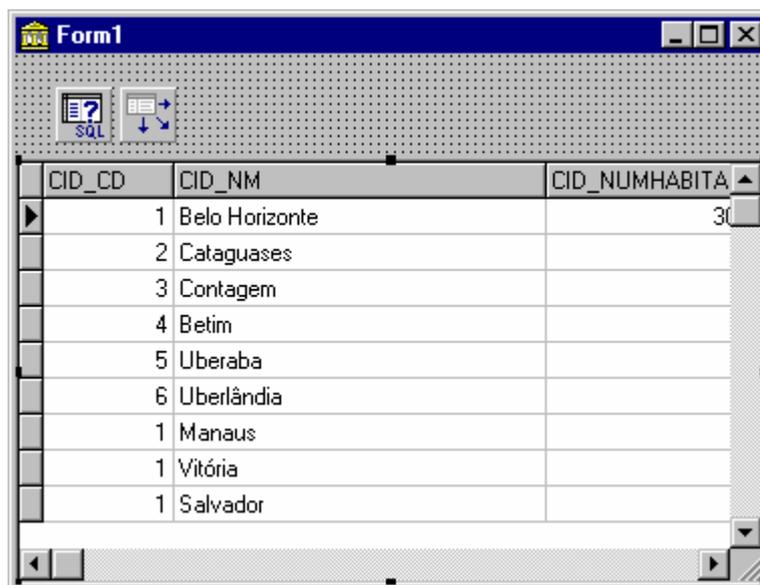


Fig 5.7: Cursor e “Result Sets” no Delphi.

Transações

Um dos recursos mais importantes fornecidos pelos bancos de dados é o controle de transação. O controle de transação permite as aplicações manterem a integridade lógica dos dados em um determinado processo. Uma transação é um conjunto de comandos executados no banco de dados que devem ser aplicados integralmente. Se um dos comandos

falhar, todos os outros têm que ser desfeitos para manter a integridade e a unicidade do processo em execução. Desta forma pode-se garantir a confiabilidade dos dados armazenados no banco. Considere o seguinte exemplo: um processo de transferência entre contas correntes que deve debitar um valor em uma das contas e creditar exatamente o mesmo valor em uma outra conta. Se o processo falhar no meio e esse recurso de controle de transação não estiver sendo utilizado, o dinheiro pode ser retirado da primeira conta, mas não ser creditado na segunda, assim o valor que deveria ser transferido, simplesmente desaparece da base de dados.

Para fazer o controle de transação, são fornecidos normalmente pelos bancos de dados três comandos: um para iniciar a transação (**begin transaction**), um para finalizar e aplicar as modificações executadas (**commit**) e outro para finalizar cancelando e desfazendo toda a operação (**rollback**). Portanto, quando uma transação é iniciada, todos os comandos seguintes fazem parte de uma única transação até ser encontrado um comando que a finalize (**commit** ou **rollback**). Se todos os comandos foram executados com êxito, pode-se então disparar o comando **commit** para que estes sejam aplicados e se tornem visíveis para os demais usuários. Caso contrário, se um erro foi encontrado em algum dos comandos, pode-se disparar um **rollback** descartando todas as alterações desde o começo da transação.

Cada conexão, normalmente suporta apenas um controle de transação e todos os seus contextos fazem parte dessa mesma transação. Quando uma transação é iniciada, qualquer comando disparado em qualquer um dos contextos faz parte da transação. E quando a transação é fechada os comandos executados por todos os contextos são aplicados e os contextos são destruídos. Sendo assim, comandos que eram mantidos preparados em algum contexto necessitam ser novamente compilados e contextos que possuíam cursores e “result sets” que não haviam sido trazidos totalmente para a aplicação são fechados e portanto seus dados perdidos. Algumas ferramentas, como o Delphi, ao verificarem que os contextos serão destruídos buscam todos os registros de “result sets” pendentes para a aplicação antes que estes sejam destruídos.

Alguns bancos de dados permitem que sejam mantidos os contextos quando a transação é finalizada. Entretanto, é necessário que a ferramenta de desenvolvimento também faça uso desse recurso. Pode-se também criar mais de uma conexão, para conseguir manter abertos os contextos, deixando-os em uma conexão separada da conexão que a transação está sendo executada, já que o controle de transação só é válido para uma única conexão. Porém, deve-se tomar bastante cuidado com essa última alternativa. Como será visto adiante, o controle de travamento de registros também é feito para cada transação e portanto uma conexão pode travar comandos da outra conexão deixando a aplicação em “DeadLock”, se os cuidados necessários não forem tomados.

Alguns servidores de banco de dados também suportam o controle de transação entre dois bancos diferentes através do recurso chamado “two-phase commit”. Esse recurso permite que a finalização da transação seja feita em dois passos, possibilitando que um banco de dados notifique ao outro o sucesso da transação para que ela possa ser finalizada como um todo.

Transações no Delphi

Para se fazer esse controle de transação no Delphi deve-se utilizar um conjunto de métodos existentes no componente *TDatabase*.

- **StartTransaction** - Inicia a transação.
- **Commit** - Efetiva as alterações feitas desde o início da transação
- **Rollback** - Cancela as alterações feitas desde o início da transação

Projeto Exemplo

O projeto a seguir mostra como as transações são tratadas nas operações feitas pelo próprio Delphi e como se comportam os cursores e contextos já abertos. Quando não se usa explicitamente os comandos vistos acima, o Delphi abre implicitamente uma transação a cada comando de atualização disparado no banco e logo a seguir fecha a transação com um comando **commit**. O objetivo do projeto é observar o que ocorre com o “Result Set” aberto, quando um comando que finaliza a transação é executado. Para construir a aplicação deve-se desenhar o “form” como mostrado na figura e definir as propriedades listadas na tabela.

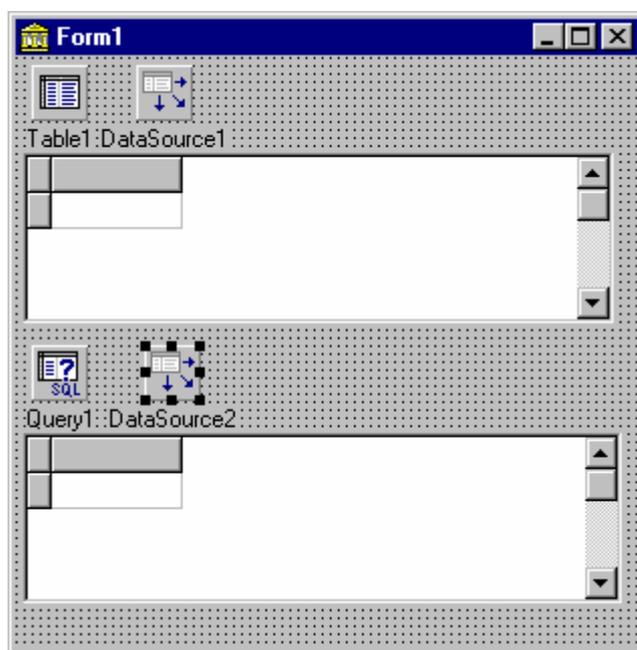


Fig 5.8: Form do projeto de transações.

Componente	Propriedade	Valor
Form1	Name	frmTransacao
Table1	DatabaseName	Vendas
	TableName	Produto
	Active	True
DataSource1	DataSet	Table1
Grid1	DataSource	DataSource1
Query1	DatabaseName	Vendas
	SQL	Select * from Produto
	Request Live	True
DataSource2	DataSet	Query1
Grid2	DataSource	DataSource2

Tabela de Propriedades: Projeto de transações.

Após ter construído o “form”, deve-se abrir o SQLMonitor através da opção do Menu **Database/SQL Monitor** para que se possa observar os comandos enviados pelo Delphi ao servidor de banco de dados.

SQL Monitor - Ferramenta capaz de monitorar as chamadas feitas pelas aplicações Delphi aos servidores de banco de dados. Permite ver as instruções enviadas para o banco como os comandos SQLs (**select, update, insert, delete**). Além disso permite ver os valores de parâmetros enviados para os comandos e os dados que são retornados pelo servidor. Concluindo, essa é uma importante ferramenta que auxilia o desenvolvedor a descobrir o comportamento da aplicação em relação ao banco de dados, tornando possível o aperfeiçoamento da aplicação para se obter uma melhor performance no ambiente cliente/servidor.

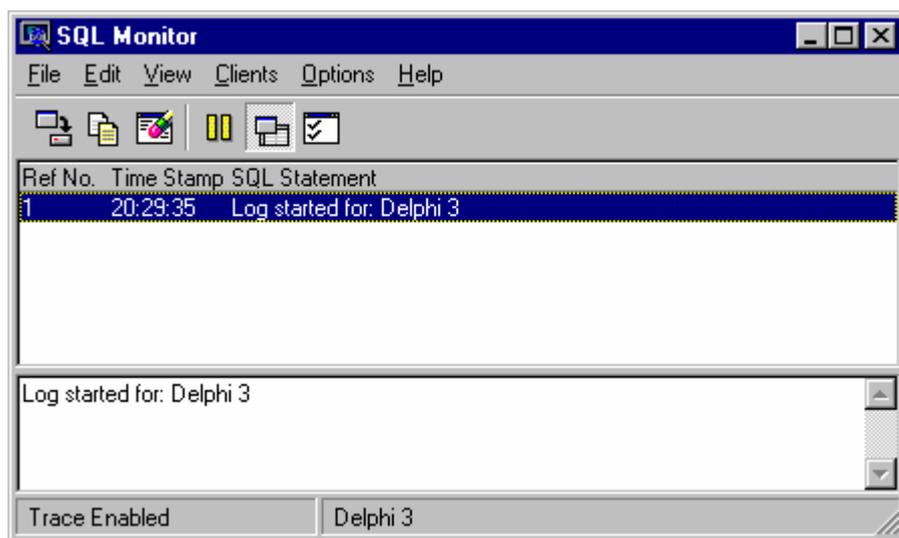


Fig 5.9: SQL Monitor.

Pode-se escolher o que será monitorado pelo SQL Monitor através da opção **Trace Option**.

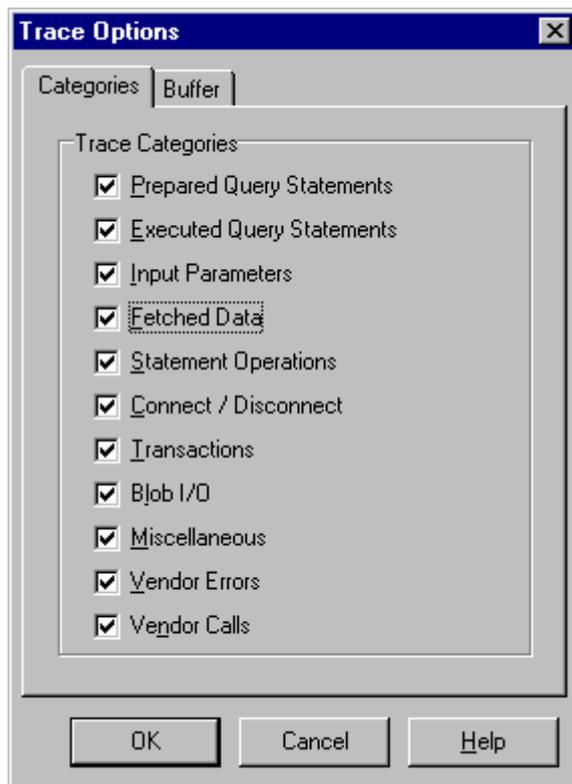


Fig 5.10: Trace Options do SQLMonitor

Após abrir o SQL Monitor deve-se executar a aplicação e observar que somente são trazidos para a aplicação uma quantidade de dados suficiente para preencher a parte visível do “Grid”. Quando o usuário navega pelo “Grid”, novas linhas são trazidas, até que o usuário atinja o fim do “result set”.

Além disso, deve-se observar que se for feita uma atualização em algum registro, será disparado um comando **close** fechando o “result set” e quando o usuário navega para uma linha que ainda não foi trazida para a aplicação, um novo comando **select** será feito buscando as linhas que ainda não foram trazidas através de um filtro colocado na cláusula **where** do comando.

```
SELECT PRO_CD ,PRO_NM ,PRO_PRECO ,PRO_ESTOQUE ,TIMESTAMP FROM PRODUTO
WHERE (PRO_CD> ?) ORDER BY PRO_CD ASC
```

Através desse exemplo, pode-se observar que quando uma transação é finalizada todos os “result sets” abertos são fechados e no caso do componente *TTable* um novo **select** é feito quando o usuário navega por registros ainda não enviados a estação.

Existe, no entanto, uma diferença quando se utiliza o componente *TQuery*. Nesse componente, quando a transação é finalizada, o Delphi lê todas as linhas pendentes até o final do “result set” antes de executar o comando **commit**. Para verificar esse comportamento, deve-se alterar a propriedade **Active** do *Table1* para “False” e a propriedade **Active** do *Query1* para “True” e fazer os mesmos procedimentos do exemplo anterior.

Porém alguns bancos suportam que os cursores permaneçam abertos quando a transação é finalizada. O Interbase é um dos bancos que possuem essa característica e para utilizá-la é necessário alterar a propriedade DRIVER FLAGS no BDE. Entretanto essa alternativa só é válida para transações controladas implicitamente pelo Delphi.

DRIVER FLAGS	Isolation level and commit type
0	Read committed, hard commit
512	Repeatable read, hard commit
4096	Read committed, soft commit
4068	Repeatable read, soft commit

Para cada servidor de banco de dados existe uma configuração diferente. Para o Oracle, por exemplo, nada precisa ser feito porque este é o comportamento padrão.

Concorrência - DLSF

Pode-se definir como concorrência o processo de disponibilizar a mesma base de dados à vários usuários conectados simultaneamente. Para manter a unicidade de uma transação e a confiabilidade no valor do dado apresentado ao usuário, é necessário a utilização de alguns controles sobre os registros do banco de dados.

Tipos de travamentos (locks)

Para disponibilizar tal funcionalidade, o banco de dados provê serviços para bloquear um registro, de forma que outros usuários não possam acessá-lo. Alguns bancos permitem o travamento de um único registro, enquanto outros trabalham com unidades compatíveis com o dispositivo físico, permitindo o travamento através de páginas. Dependendo do banco, o tamanho dessas páginas pode variar (2K, 1K), podendo essas serem formadas por mais de um registro. Portanto, quando uma página é travada, todos os registros que ela possui serão travados. Esse controle de atribuir e remover as travas é realizado para cada transação em cada conexão. Mesmo se duas conexões são feitas pelo mesmo usuário, o controle de travamento é distinto, podendo assim uma das conexões bloquear registros para a outra conexão. Deve-se ter cuidado ao se trabalhar em mais de uma conexão para não acontecer os chamados “Deadlocks”, que são travamentos causados por mais de uma conexão e que cada uma delas está a espera da liberação de um registro da outra para continuar sua execução.

Pode-se dizer que os bancos de dados trabalham com dois tipos de travas: “exclusive lock” e “shared lock”.

Exclusive Lock

São travas atribuídas aos registros ou páginas quando o usuário executa comandos de atualização sobre elas (**update, insert, delete**). Como seu nome indica, nenhuma outra trava pode ser colocada na mesma página ou registro, proibindo qualquer tipo de acesso ao registro ou página por outra conexão (transação). Todas as travas “exclusive locks” são retiradas automaticamente quando a transação for finalizada por um **commit** ou **rollback**.

Shared Lock

São travas que podem ser colocadas nos registros ou páginas quando o usuário seleciona o registro. Uma trava “shared lock” permite que outras travas do mesmo tipo sejam atribuídas ao mesmo registro ou página, entretanto proíbe a atribuição de uma trava “exclusive lock”. Portanto, quando se atribui a um registro ou página uma trava “shared lock”, bloqueia-se qualquer atualização por outra transação (conexão), permitindo assim a aplicação ter acesso ao dado com a certeza que este não será alterado por mais ninguém. Entretanto, é possível que o mesmo usuário na mesma conexão possa alterar o registro ou página marcado por ele

como “shared lock” transformando-o para “exclusive lock”, desde que nenhuma outra conexão possua também uma trava “shared lock” para o mesmo registro ou página. Todas as travas “shared lock” também são retiradas quando se finaliza a transação.

Pág. ou Regist.\Transações	1	2	3	4	5
1	Exclusive				
2	Shared	Shared			
3		Shared	Shared	Shared	
4				Exclusive	
5					

Tabela de Transações: Um “exclusive lock” em uma linha, exclui a possibilidade de existir outro “exclusive lock” ou “shared lock” em uma outra coluna na mesma linha.

Níveis de isolamento

Podemos definir alguns níveis de isolamento, de acordo com a forma que os “shared locks” são atribuídos aos registros ou páginas. A seguir serão mostradas as várias formas de se trabalhar com “locks” no banco de dados, independente dos nomes utilizados pelos vários fornecedores. Sendo possível que alguns bancos de dados não apresentem todas as possibilidades descritas abaixo.

Travamento de todos os registros

Nessa modalidade é atribuída uma trava “shared lock” para todos os registros da seleção feita através de um comando **select**. Este é o nível de isolamento que mais trava os registros e deve ser utilizado com cautela para não inviabilizar a concorrência dos dados e o funcionamento da aplicação.

Travamento de um registro ou página

Nesse tipo de travamento, são colocadas travas “shared locks” somente para o registro ou página onde o cursor se encontra posicionado. Quando a aplicação movimentar o cursor para um próximo registro, é retirada a trava do registro anterior e colocada no registro atual. Portanto, somente um registro ou página fica travado em um determinado momento.

Sem travamento

Nessa modalidade é atribuída uma trava “shared lock” no momento de acesso a cada registro. Entretanto, essa trava é retirada logo a seguir e o registro ou página não permanece travado. Essa pequena atribuição da trava é simplesmente para verificar se não existe nenhuma outra trava do tipo “exclusive lock” sobre o registro, que não permitiria a sua leitura. Portanto, mesmo nessa categoria não é possível acessar registros alterados no meio de uma transação que estão marcados como “exclusive lock”.

Seleção de registros travados com um exclusive lock.

Alguns bancos permitem que uma conexão acesse os dados de um registro ou página mesmo se estes estiverem marcados como “exclusive lock”. Para tal, o registro é acessado sem a atribuição de nenhuma trava, já que uma trava “SL” não pode coexistir com uma trava “EL” em um mesmo registro ou página. Como os registros acessados possuem dados que não foram efetivados através de um **commit**, são normalmente disponibilizadas cópias antigas do mesmo registro armazenadas pelo banco de dados. Apesar de permitir esse tipo de acesso, muitos bancos de dados proíbem a alteração dos dados quando acessados desta forma.

Optimistic Lock

Para se maximizar a concorrência entre os dados nas aplicações, podem ser utilizadas outras formas de controle que diminuem os travamentos causados na base de dados. Para isso pode-se utilizar o nível de isolamento “Sem travamentos” e fazer um controle adicional para ter certeza que os dados mostrados para o usuário não foram alterados quando esse inicia o processo de alteração do registro.

Existem algumas alternativas que podem ser seguidas para se implementar esse processo. O processo normal de um comando de **update** ou **delete** é conter na cláusula **where** a verificação da chave primária para possibilitar a identificação do registro na tabela. Entretanto, explorando um pouco mais esse recurso de identificação do registro, pode-se implementar um bom controle de concorrência.

Where All

Consiste em verificar na cláusula **where** não só a chave primária, mas todos os registros. Assim, se qualquer um dos registros for alterado ou excluído por outro usuário, esse não será encontrado quando o comando **update** ou **delete** for disparado, sendo possível a aplicação notificar o usuário tal acontecimento. A aplicação pode então permitir o usuário buscar novamente os dados para verificar as alterações feitas e reiniciar o processo de alteração.

Entretanto, campos de difícil comparação, como campos contendo imagens e textos grandes, normalmente não podem ou não devem ser utilizados desta forma.

Where Changed/Key

Consiste em verificar na cláusula **where** a chave primária mais os campos que sofreram alteração pelo usuário. Desta forma garante-se que os mesmos campos não foram alterados por outro usuário. Mas, permite atualizar um registro alterado por outro usuário desde que não sejam os mesmos campos.

Where Key

Nesse método a concorrência é livre, pois permite que o registro seja alterado sem verificar se este foi alterado por outro usuário. Na cláusula **where** de alteração, é somente utilizada a chave primária da tabela para localizar o registro.

Where Timestamp/Rowid

Consiste em verificar na Cláusula **where** a chave primária mais uma coluna definida exclusivamente para fazer o controle de concorrência. Essa coluna é atualizada para um novo valor toda vez que o registro for inserido ou alterado. Portanto, se o registro for alterado por um outro usuário, o registro não será encontrado para atualização.

Alguns tipos de bancos de dados já possuem algum suporte para esse tipo de controle. O SQLServer e o Sybase possuem um tipo de dado chamado **timestamp**, para o qual pode se criar uma coluna que será atualizada automaticamente com a hora, minuto, segundo e décimos de segundo do momento em que ocorrer uma operação de inserção ou alteração. O SQLBase é um outro banco que permite esse controle através de uma coluna chamada **rowid** que já é definida automaticamente para todas as tabelas e é atualizada para um valor único quando ocorre as operações **insert** e **update**.

Para os bancos que não possuem automaticamente esses recursos, deve-se criar uma nova coluna que pode armazenar a Data/hora como no caso do **timestamp**, desde que o tipo permita a atribuição de décimos de segundo, porque a unidade segundo não possui precisão

suficiente para essa operação. Ou pode-se criar uma coluna numérica, cujo o valor será incrementado através de um contador único a cada atualização realizada sobre o registro.

Além de criar a coluna, é preciso fazer também o processo de atualização dos dados quando uma operação de **insert** ou **update** ocorrer. Pode-se fazer isso através da aplicação ou através do recurso de “triggers” dos bancos de dados. A utilização de “triggers” possibilita que a atualização possa ser feita através de outras ferramentas que não sejam a própria aplicação e é portanto mais recomendado.

Concorrência no Delphi

Para se configurar os níveis de isolamento utilizados pelo Delphi deve-se utilizar a propriedade **TransIsolation** do componente *TDatabase*.

- **tiDirtyRead** - Permite ler alterações feitas por outro usuário (ou transação) ainda não “comitadas” (efetivadas), ou seja não respeitando os “exclusives locks”. Essa opção normalmente não está disponível nos banco de dados.
- **tiReadCommitted** - Esse é o tipo mais normal de se trabalhar. Somente as alterações feitas por outro usuário (ou transação) já “comitadas” são vistas pelo usuário. Se a alteração ainda não foi efetivada o registro fica travado e a leitura fica esperando o fim da transação.
- **tiRepeatableRead** - Esse método é utilizado para que o dado visto pelo usuário seja sempre o mesmo durante a transação se este for lido novamente. Alguns bancos implementam este recurso permitindo ver uma cópia mais antiga de um registro alterado por outra transação e ainda não “comitado”. Desta forma a aplicação não fica travada esperando a liberação do registro pela transação. Esta alternativa diminui a possibilidade de travamentos entre as transações, porém consome um número maior de recursos do banco para fazer as cópias (versões) dos registros. Outros bancos, implementam este recurso travando com um “shared lock” todos os registros lidos, não permitindo assim que outro usuário altere o registro, desta forma se os dados forem relidos seus valores permanecem o mesmo. Ao contrário da implementação anterior, essa alternativa aumenta a possibilidade de travamentos entre as transações e deve ser utilizada com bastante cuidado.

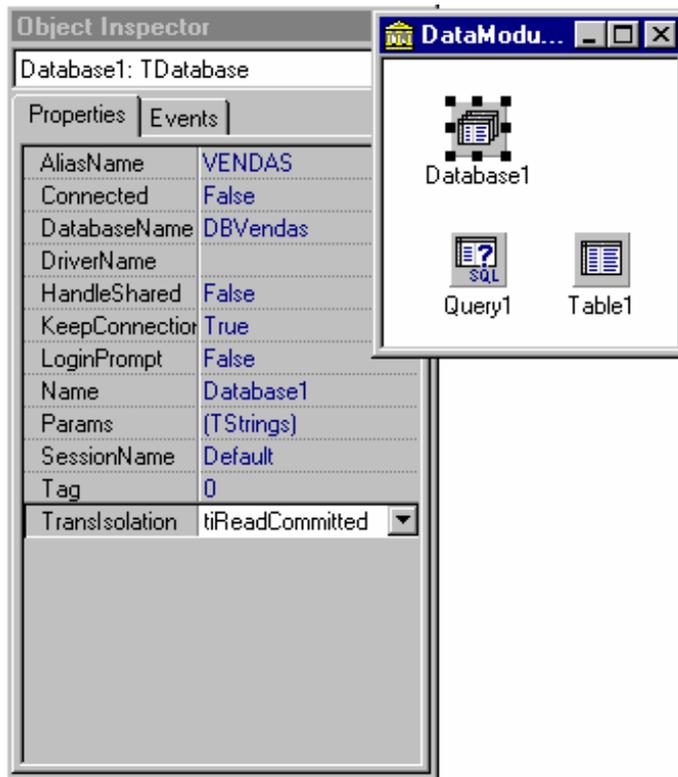


Fig 5.11: TDatabase.TransIsolation.

Para se configurar o processo de “Optimistic Lock” no Delphi, deve-se utilizar a propriedade **UpdateMode** dos componentes *DataSets*: *TQuery* e *TTable*.

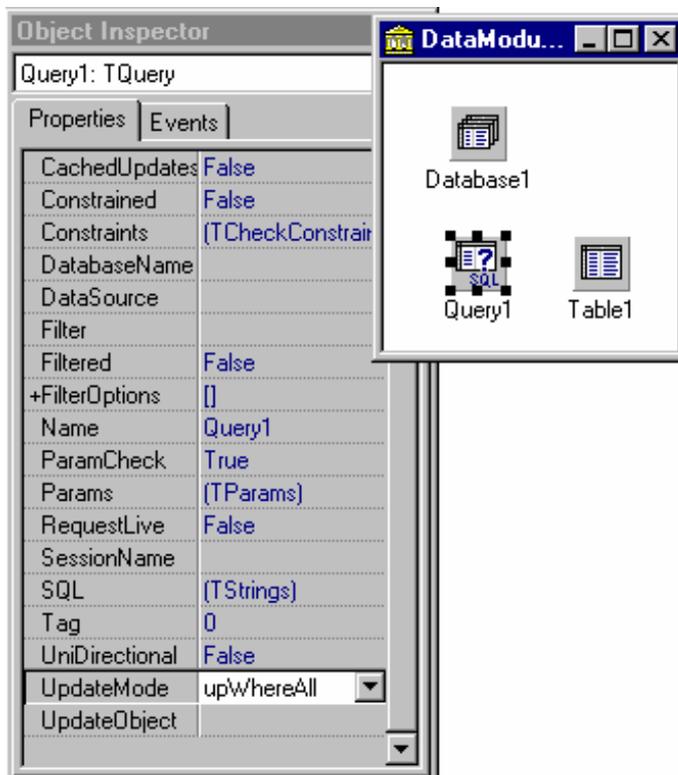


Fig 5.12: TQuery.UpdateMode.

Projeto Exemplo

O projeto a seguir mostra como esses recursos são utilizados no Delphi. O formulário apresentado na figura define como a aplicação deve ser construída. Deve-se utilizar duas instâncias da aplicação para simular o acesso de dois usuários simultaneamente. O exemplo mostra o processo de travamento do registro quando uma atualização é feita por um usuário antes do comando **commit** ser executado.

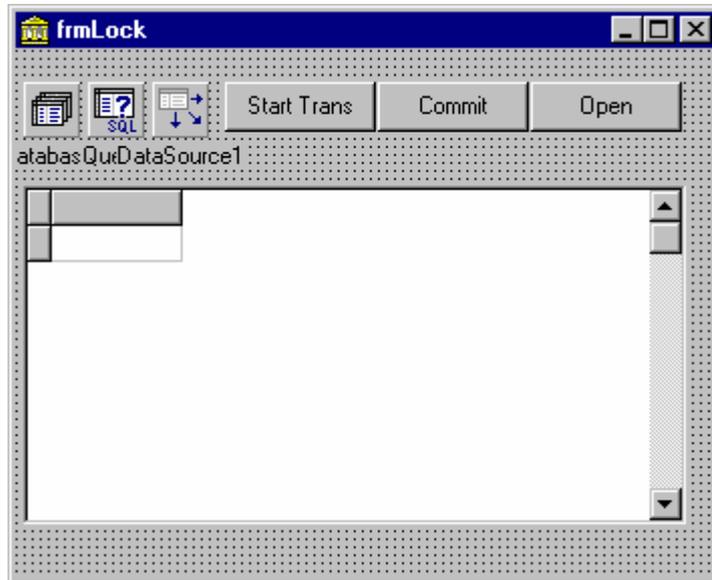


Fig 5.13: Projeto Exemplo

Componente	Propriedade	Valor
Form1	Name	frmLock
Database1	Alias	VENDAS
	DatabaseName	DBVENDAS
	Connected	True
Query1	DatabaseName	DBVENDAS
	SQL	Select * from Cidade
	Request Live	True
DataSource1	DataSet	Query1
Grid1	DataSource	DataSource1
Button1	Caption	Start Trans
Button2	Caption	Commit
Button3	Caption	Open

Tabela de Propriedades.

O botão “Start Trans” será utilizado para iniciar a transação e portanto é implementado da seguinte maneira:

```

procedure TfrmLock.Button1Click(Sender: TObject);
begin
    Database1.StartTransaction;
end;
    
```

O botão “Commit” ao ser pressionado fechará a transação executando o método **commit** do *TDatabase*:

```

procedure TfrmLock.Button2Click(Sender: TObject);
    
```

```
begin
  Database1.Commit;
end;
```

E o botão “Open” deve alternar entre abrir e fechar o *DataSet Query1*.

```
procedure TfrmLock.Button3Click(Sender: TObject);
begin
  Query1.Active:= Not Query1.Active;
end;
```

Depois de montada a aplicação deve-se compilá-la, executá-la e abrir mais uma instância executando o EXE através do windows explorer.

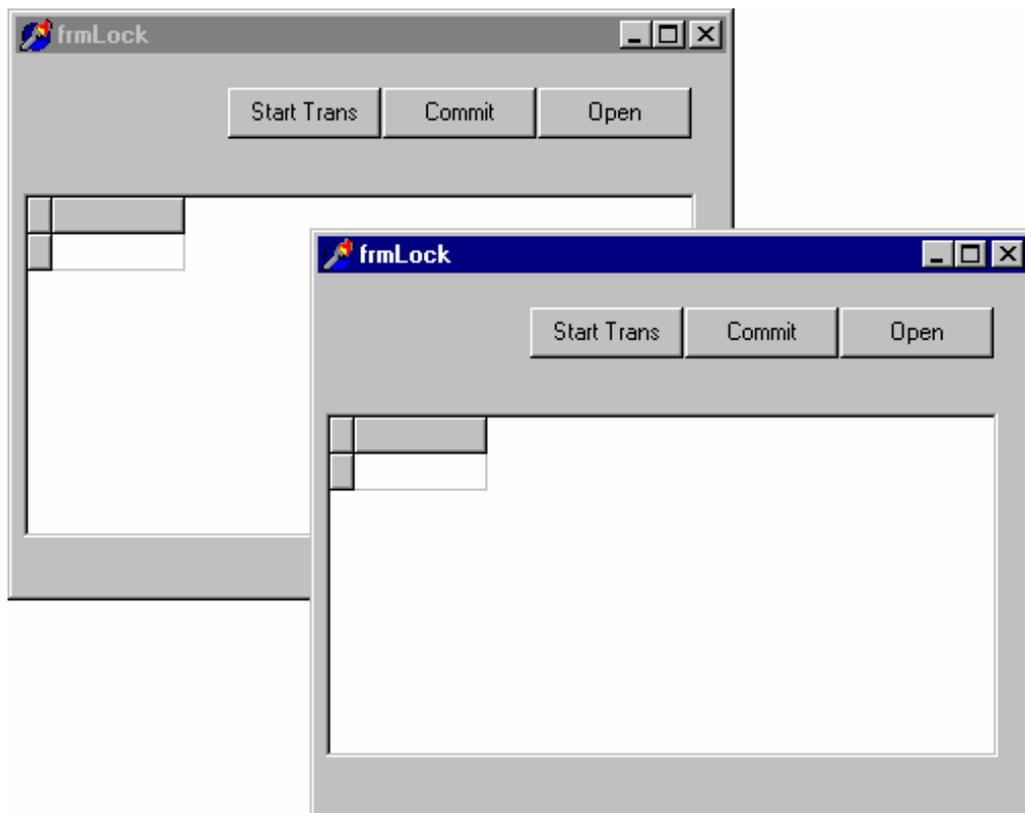


Fig 5.14: Duas instâncias do projeto exemplo em execução.

Deve-se executar agora os seguintes passos:

1. Pressionar o botão “Open” da primeira instância para abrir o *DataSet*;
2. Pressionar o botão “Start Trans” da mesma instância para iniciar a transação;
3. Alterar o campo CID_NUMHABITANTES do primeiro registro para 400.000;
4. Navegar para o segundo registro para que a gravação seja feita no banco;
5. Tentar abrir a “query” na segunda instância da aplicação através do botão “Open”. O “result set” virá vazio, tratamento feito pelo Delphi quando encontra o registro bloqueado;

6. Pressionar o botão “commit” da primeira instância para efetivar a gravação e liberar os “locks”;
7. Fechar e abrir novamente a “query” da segunda instância através do botão “Open”. Agora, os registros são trazidos para a tela;
8. O mesmo acontece se o usuário da segunda instância tentar alterar os dados que ainda não foram efetivados. Pressionar novamente o botão “Start Trans” da primeira instância e alterar o campo CID_NUMHABITANTES para 300.000;
9. Tentar, agora, alterar o mesmo registro na segunda instância. Uma mensagem de “DeadLock” é retornada informando que o registro se encontra travado. Isso acontece, porque quando se tenta alterar o registro da segunda instância o Delphi tenta atualizar o registro, mas esse se encontra bloqueado e não pode ser lido;



Fig 5.15: DeadLock.

Nesse exemplo foi utilizado o nível de isolamento “ReadCommitted”, que somente permite o usuário lê registros que já foram efetivados por outro usuário. Pode-se fazer o mesmo exemplo anterior alterando antes o nível de isolamento no componente *Database1* para “RepeatableRead”;



Fig. 5.16: Propriedade TransIsolation do componente Database1

Entretanto, esse nível só é atribuído quando se inicia a transação. Portanto, antes de executar o passo 5, que abre a “query”, deve-se abrir a transação na segunda instância da aplicação

para definir o novo nível de isolamento. Assim, os dados são lidos mesmo antes do primeiro usuário efetuar o **commit**, mas com os valores anteriores a alteração feita por ele.

Podemos utilizar esse exemplo para verificar também o comportamento da propriedade **UpdateMode** do componente *Query1*.

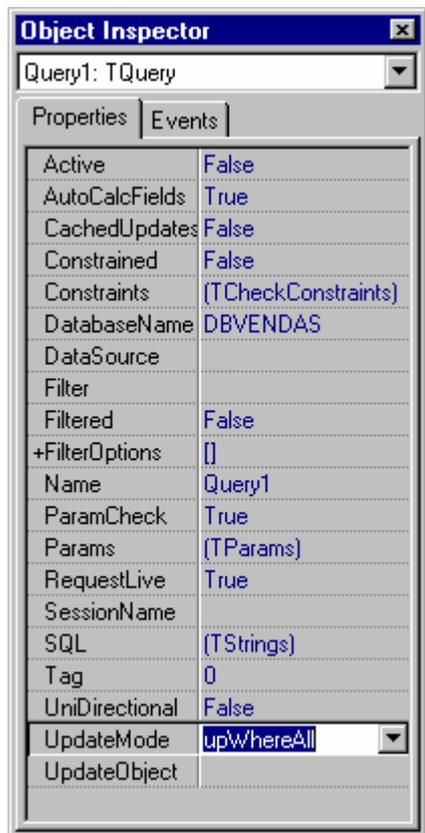


Fig. 5.17: Propriedade **UpdateMode** do componente *Query1*.

O objetivo desse exemplo foi monitorar o acesso ao banco de dados através da alternância de várias propriedades dos componentes de uma aplicação Delphi, de forma a verificar as diversas possibilidades que o desenvolvedor pode optar durante a construção da aplicação.

Em uma aplicação real, não se separa dessa forma os comandos de abrir uma transação, executar a operação e fechar a transação. Essas operações devem ser executadas seqüencialmente em uma única interação com o usuário em um período de tempo bem pequeno. Não cabe ao usuário tomar a decisão de abrir e fechar a transação, porque ele poderia deixar travado durante muito tempo um registro que vários outros usuários precisam utilizar.

Projetando Aplicações Cliente/Servidor

Esse capítulo mostra algumas técnicas de construção de aplicações voltadas para o ambiente Cliente/Servidor.

A construção de aplicações Cliente/Servidor deve obedecer algumas regras de organização para que os sistemas possam ter um tempo de vida mais longo e acompanhar a evolução do negócio da empresa e a evolução tecnológica da arquitetura.

Estrutura de uma Aplicação

Como foi visto, o ambiente cliente servidor permite que a aplicação seja distribuída entre dois componentes físicos: a estação cliente e o servidor de banco de dados. Entretanto, logicamente podemos identificar três camadas distintas dentro de uma aplicação.

Apresentação

Composta por componentes responsáveis pela interação da aplicação com o usuário final. É responsabilidade dessa camada receber os dados e comandos do usuário e devolver-lhe informações através de elementos visuais como consultas, gráficos, relatórios e etc;

Lógica do Negócio

Parte da aplicação responsável por manter as regras de negócio da empresa. Essa camada recebe os dados da camada de interface e executa as operações e validações necessárias para enviá-los ao banco de dados. Da mesma forma, extrai os dados do banco de dados de acordo com as regras de negócio da aplicação e os envia para elementos da interface para que sejam exibidos.

Portanto, essa camada é responsável em interligar a interface visual com o banco de dados através da execução de transações, consistência dos dados e regras de negócio, ou seja, a parte funcional da aplicação.

Gerenciamento de Dados

Parte da aplicação responsável pelo acesso e a manipulação dos dados no servidor. Como já foi visto anteriormente, grande parte dessa camada é implementada pelo próprio servidor de banco de dados.

Normalmente o acesso aos serviços é feito através da linguagem SQL. Porém, é também necessário um conjunto de comandos para enviar as sentenças SQLs e gerenciar a comunicação entre a aplicação e o servidor. Esses comandos se encontram em bibliotecas disponibilizadas pelos próprios fornecedores de banco de dados que são instaladas em cada estação de trabalho. Além disso cada fabricante de ferramentas de desenvolvimento fornece também métodos e componentes capazes de simplificar e tornar mais transparente o acesso aos diversos SGDBs.

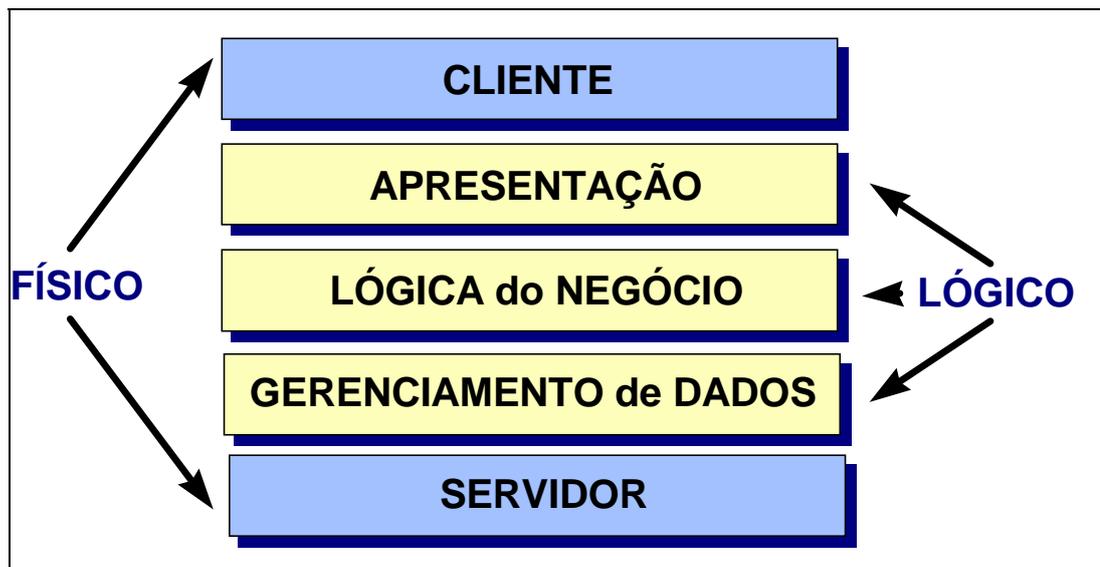


Fig 6.1: Camadas Físicas e Lógicas de uma Aplicação.

Vantagens da Organização da Aplicação em Camadas

A divisão da aplicação nessas três camadas lógicas possibilita a organização e padronização da codificação e construção da aplicação, além de proporcionar uma maior facilidade de manutenção e evolução para novas fases da arquitetura Cliente/Servidor. Como já foi visto, a tendência da arquitetura Cliente/Servidor é retirar cada vez mais parte do processamento da aplicação realizado pelas estações de trabalho clientes e centralizá-lo em servidores, provendo um melhor gerenciamento do processo e facilitando a evolução da funcionalidade que foi distribuída.

A distribuição da aplicação em camadas lógicas possibilita também que cada camada possa evoluir independente das outras desde que se mantenha a interface entre elas. Por exemplo, pode-se alterar as regras de negócio para atender as necessidades do mercado sem necessariamente ter que modificar a camada de interface ou a camada de gerenciamento de dados. Por outro lado, pode-se evoluir a apresentação para novas tecnologias como multimídia, sem precisar alterar as regras de negócio. Evoluções tecnológicas, como a

distribuição da base de dados na camada de gerenciamento de dados pode ser feita de forma transparente das demais camadas. Portanto, esses tipos de distribuição tornam as aplicações mais escaláveis para suportar futuras implementações possibilitando um tempo de vida muito mais longo.

Estrutura de uma Aplicação Delphi

A ferramenta de desenvolvimento Delphi, desde sua primeira versão já se mostrou preocupada em distribuir a aplicação nas três camadas lógicas. Esse esforço inicial tem beneficiado a Borland a evoluir a ferramenta para as novas gerações da arquitetura Cliente/Servidor. Muitas outras ferramentas que não possuíam essa filosofia de trabalho estão enfrentando sérios problemas para sensibilizar seus clientes a necessidade de trabalhar dessa forma para conseguirem migrar e receber os benefícios da nova geração Cliente/Servidor.

Para implementar essa filosofia de trabalho no Delphi, a Borland estabeleceu três categorias de componentes: componentes visuais, componentes de acesso à base de dados e componentes de ligação.

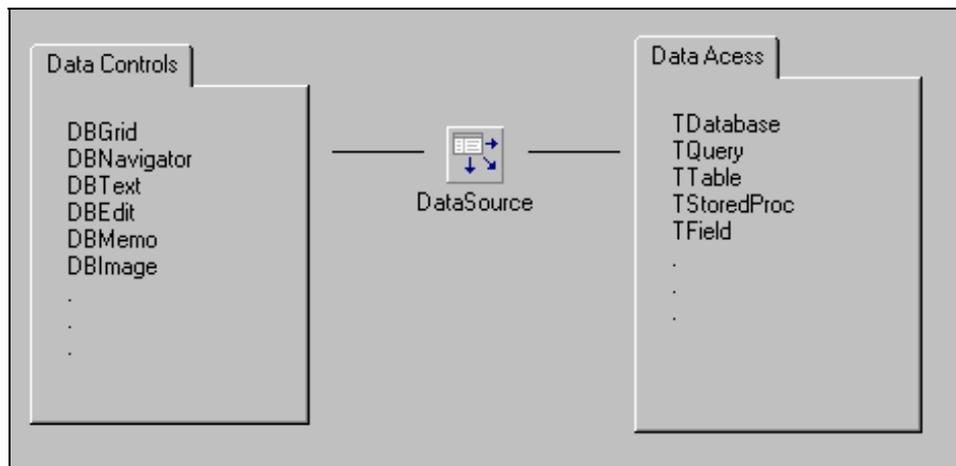


Fig 6.2: Categorias de componentes do Delphi para acesso a base de dados.

Essa forma de trabalho organizada em camadas distintas permite também uma maior reutilização de código e portanto um aumento de produtividade na construção de aplicações através dos recursos do Delphi de criação de componentes e “templates” de telas.

Componentes visuais

Componentes responsáveis pela interface com o usuário. Correspondem à camada de Apresentação discutida anteriormente. Esses componentes estão localizados na página **Data Controls** da paleta de componentes do Delphi.

Construindo Aplicações Cliente/Servidor

Esse capítulo mostra a construção de uma tela de manutenção utilizando-se os componentes do Delphi e como configurá-los para trabalhar com banco de dados.

Existem várias maneiras de se utilizar os componentes do Delphi para trabalhar com banco de dados. É preciso saber escolher as formas mais adequadas para cada tipo de implementação. Através desse capítulo serão mostradas várias dessas opções e o que cada uma difere no comportamento da aplicação com o banco de dados.

Utilizando Data Modules

Em uma aplicação Cliente/Servidor é interessante utilizar o recurso de Data Module fornecido pelo Delphi. Data Module é um componente que fornece uma localização centralizada para componentes não-visíveis do Delphi. Pode-se então utilizar o Data Module para conter os componentes da categoria Data Access, responsáveis pela lógica de negócio da aplicação e interface com o banco de dados. Assim, toda a lógica de negócio fica concentrada em um único ponto da aplicação facilitando a manutenção e evolução. Para aplicações maiores é normal que se utilize mais de um Data Module subdividindo a lógica de negócio.

Para exemplificarmos esse recurso, iremos construir a tela de cadastro de produtos.. Para se construir essa tela, deve-se inicialmente criar um *DataModule* e colocar nele um componente *Database*.



Fig 7.1: Componente *TDatabase*.

Componente	Propriedade	Valor
DataModule1	Name	DMSistVendas

Tabela de Propriedades.

Componente TDatabase

Como foi visto o componente *TDatabase* é responsável pela conexão com o banco de dados. Esse componente possui algumas propriedades que permitem a configuração dessa conexão. Pode-se utilizar a opção **Database Editor** do popup menu que é exibido ao se pressionar o botão direito do mouse sobre o componente *Database1*.

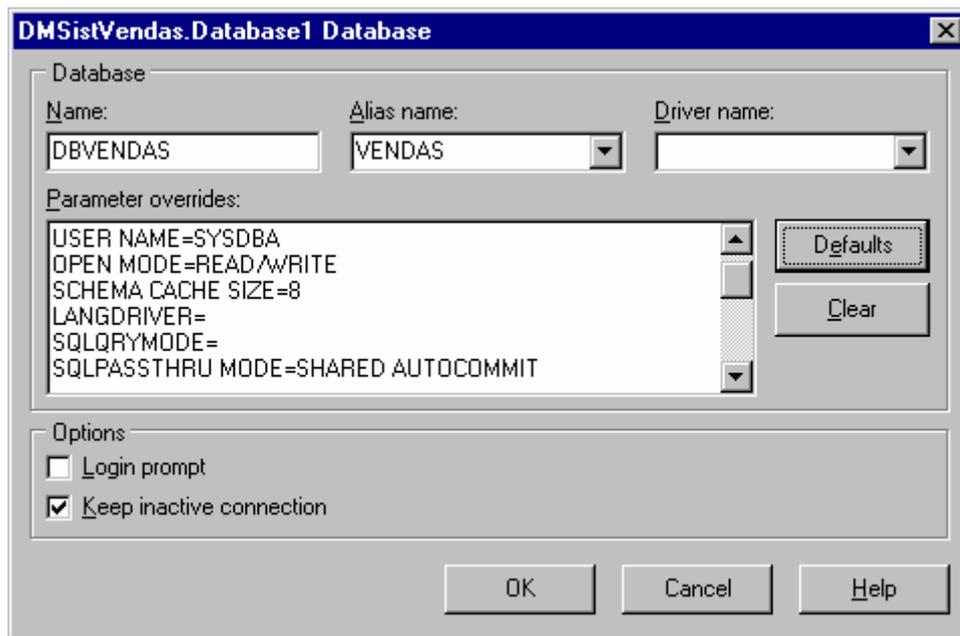


Fig 7.2: Database Editor

Através do Database Editor pode-se atribuir os valores para várias propriedades como foi mostrado na figura. Além disso, sobrepor valores do “Alias” definidos no BDE através do botão **Defaults**.

- **USER NAME:** Esse parâmetro pode ser utilizado para deixar fixo o usuário que irá acessar o banco de dados. Algumas empresas adotam essa forma de trabalho ao invés de criar vários usuários no banco de dados. Em nosso exemplo, iremos atribuir o valor SYSDBA para o parâmetro.
- **PASSWORD:** Se for fixado o valor de USER NAME, pode-se também já deixar especificada a senha do usuário. PASSWORD=masterkey
- **SQLPASSTHRU MODE:** Esse parâmetro possui três valores possíveis: SHARED AUTOCOMMIT, SHARED NOAUTOCOMMIT e NOT SHARED. Deve-se especificar SHARED, quando deseja-se que todas as atualizações feitas automaticamente pelo Delphi, as atualizações feitas manualmente e o controle de transação usem a mesma conexão. Quando a opção NOT SHARED é utilizada, uma nova conexão escondida é

estabelecida com o banco de dados. Assim, todas as atualizações feitas manualmente através do método **ExecSql** de uma *Query* e o controle de transação feito através dos métodos do componente *TDatabase* utilizam essa nova conexão. Além disso, se especificada a opção **SHARED**, pode-se ainda escolher entre as opções **AUTO COMMIT** e **NOAUTO COMMIT**. A primeira efetiva automaticamente qualquer comando SQL enviado ao servidor, a menos que uma transação seja aberta explicitamente através do comando **start transaction**. Na segunda opção, os comandos esperam pela execução explícita do comando **commit** para serem efetivados. Em nosso exemplo, iremos atribuir o valor **SHARED AUTO COMMIT** para o parâmetro.

Escolhendo entre TTable e TQuery

O Delphi possui dois componentes que permitem o acesso e manipulação dos dados do servidor: *TTable* e *TQuery*. O primeiro é baseado no acesso a uma determinada tabela do banco de dados e o segundo é baseado em uma sentença SQL (comando **select**). Por ser baseado em uma sentença SQL, o componente *TQuery* permite trabalhar com mais de uma tabela do banco de dados ao mesmo tempo. Ambos os componentes utilizam a linguagem SQL para acessar a base de dados. Quando se trabalha com *TTable*, o Delphi gera automaticamente uma sentença SQL de acordo com os parâmetros definidos para o componente. Além dessa diferença básica entre os dois componentes, outros fatores devem ser observados na escolha.

Abertura

Operação feita quando é executado o método **Open** do componente *TTable* ou *TQuery*, que produz a compilação e execução do comando **select**. Quando esse método é executado através do componente *TTable*, o Delphi realiza uma série de outros comandos SQLs para buscar informações do catálogo da tabela necessárias para as operações de seleção e atualização. Essa busca pode ser otimizada através da opção **ENABLE SCHEMA CACHE** do BDE, fazendo com que essas informações sejam lidas apenas uma vez durante a execução da aplicação. Quando o primeiro acesso é feito, o BDE armazena as informações em um arquivo e qualquer nova necessidade de abertura da mesma tabela não necessita buscar novamente os elementos do catálogo.

Por outro lado, utilizando-se o componente *TQuery*, pode-se desviar dessa busca desde que não se utilize a propriedade **Request Live** que torna o “result set” da “query” atualizável automaticamente pelo Delphi. Se o valor da propriedade **Request Live** for **TRUE** e o **SELECT** utilizado obedecer as restrições para que o Delphi consiga atualizar o “result set”, as mesmas buscas utilizadas para o componente *TTable* terão que ser feitas.

Concluindo, para que a busca de elementos do catálogo não seja feita é necessário utilizar o componente *TQuery* e controlar as atualizações manualmente ou através de componentes do tipo *TUpdateSQL*.

Filtros

Uma das grandes vantagens de se utilizar SGBDs é poder fazer o filtro no próprio servidor e portanto trafegar um número menor de linhas pela rede. Para se fazer isso, é necessário utilizar a cláusula **where** do comando **select**. Quando se envia uma cláusula **where** no comando SQL, o próprio servidor se encarrega de selecionar os registros que compõem a

pesquisa realizada, já observando as melhores alternativas de acesso tentando utilizar o máximo dos índices estabelecidos no banco de dados.

Com o componente *TQuery* isso é feito diretamente no comando SQL suportando a sintaxe fornecida pela linguagem SQL do banco de dados utilizado. Entretanto, se for utilizada a propriedade **filter** do *TQuery* para filtrar o “result set”, o Delphi não utilizará os recursos do SGBD para selecionar os registros e irá trazer todas as linhas resultantes da Query para a estação. Somente quando essas linhas forem trazidas para máquina cliente, é que o filtro será aplicado localmente, tornando cada linha visível ou não na tela.

Entretanto, com o componente *TTable* a propriedade **filter** e as funções de seleção como **SetRange** agem de forma diferente. O Delphi tenta traduzir as especificações feitas através desses dois métodos e colocá-las diretamente na cláusula **where** do **select** realizado. Desta forma, consegue-se o mesmo desempenho do componente *TQuery*, já que o filtro é feito na própria cláusula **where**. Entretanto, como o Delphi é que realiza a tradução das especificações para a cláusula **where**, existe uma certa limitação dessas especificações e se essas não conseguirem ser traduzidas, o filtro será feito na própria máquina cliente. Portanto, o componente *TQuery* é mais abrangente no que se diz respeito a filtros, suportando de forma mais completa a sintaxe fornecida pelo banco de dados.

Com relação ao evento **OnFilterRecord**, em ambos os componentes o filtro é aplicado localmente e portanto todas as linhas que compõem o “result set” precisam ser trazidas para a estação cliente, não utilizando os recursos do servidor.

Transações

Como já foi visto o componente *TTable* possui uma forma mais inteligente de se comportar do que o componente *TQuery* quando o “result set” está prestes a ser destruído através da realização de um comando **commit** para finalizar a transação.

O componente *TQuery* necessita que todas as linhas até o final da seleção sejam trazidas para a estação cliente antes que o **commit** seja executado no banco para que o usuário não perca as linhas que ainda não foram trazidas para aplicação.

Já o componente *TTable* simplesmente fecha o “result set” sem nenhum efeito que diminua o desempenho da atualização e se houver a necessidade de buscar as linhas restantes da tabela um novo **select** é feito a partir da última linha trazida.

Entretanto, alguns bancos de dados permitem que o **commit** não destrua os “result set” ou pode-se também utilizar conexões separadas para a atualização e para o “result set”. Desta forma não há necessidade do componente *TQuery* realizar as buscas antes da real necessidade do usuário.

Por outro lado, o componente *TTable*, deixa aberta uma transação que pode, dependendo do banco de dados, estar travando alguma página da tabela.

Número de Tabelas Acessadas

Um outro fator relevante na escolha do componente, é o número de tabelas que devem ser acessadas para buscar as informações necessárias para o usuário em uma mesma tela. Até um certo número de tabelas é mais interessante utilizar o recurso de “joins” dos bancos para trazer em um único comando SQL, todo o conjunto de informações. Nesse caso um

componente *TQuery* deveria ser utilizado. Quando isso é feito através de vários componentes *TTable*, é às vezes necessário trazer os dados de todas as tabelas para a máquina cliente para que a relação entre elas possa ser feita. No melhor caso, se filtrarmos cada tabela pelo registro selecionado na outra, teríamos que executar vários comandos `SELECT`'s mais simples no servidor contra um único comando um pouco mais complexo do componente *TQuery*.

Trabalhando com o TQuery

O componente *TQuery* pode ser então utilizado para acessar e manipular os dados. Como foi visto, a utilização desse componente deixa mais flexível o acesso ao banco, já que trabalha diretamente com a linguagem SQL. Portanto, manutenções evolutivas como acessar mais de uma tabela através do mesmo componente podem ser implementadas com mais facilidade.

Afim de continuarmos nossa aplicação, devemos colocar um componente *TQuery* no *DataModule DMSistVendas*.



Fig 7.3: Componente *TQuery*

Componente	Propriedade	Valor
Query1	Name	QProduto
	DatabaseName	DBVENDAS
	SQL	Select * from produto order by produto.prod_cd
	Request Live	TRUE
	Active	TRUE

Tabela de Propriedades.

A propriedade **Request Live** ligada, faz com que o Delphi tente atualizar automaticamente o “result set” trazido pela “query”. Desta forma, o componente *TQuery* se comporta de forma semelhante ao componente *TTable*.

Entretanto, para que isso seja possível, o comando SQL tem que obedecer algumas restrições como por exemplo:

- Conter apenas uma tabela na cláusula **From**;
- Não possuir agregações através de **group by** e funções como **sum**, **max**, etc;

Após termos preenchido o *DataModule* com os componentes responsáveis pelo acesso ao banco de dados, podemos construir o “form” de manutenção da tabela de produto como a figura a seguir:



Fig 7.4: Form de Produto.

Componente	Propriedade	Valor
Form1	Name	frmProduto
	Caption	Produtos
Panel1	Align	alTop
	Caption	
Panel2	Align	alClient
	Caption	
DataSource1	Name	DSMain
	DataSet	DMSistVendas.Qproduto
DBNavigator1	DataSource	DSMain
DBEdit1..DBEdit4	DataSource	DSMain
	DataField	pro_cd..pro_estoque

Tabela de Propriedades.

Podemos então executar a aplicação e verificar que através da propriedade **Result Live** do *Qproduto*, foi possível atualizar os dados e acrescentar novas linhas à tabela. Entretanto, podemos notar que as linhas inseridas desaparecem do “Result Set” ao navegarmos para outro registro. Esta é uma restrição que existe no componente *TQuery* quando se utiliza a propriedade **Result Live** para torná-lo atualizável automaticamente pelo Delphi. Além disso, foi visto que para ser possível tornar o “result set” atualizável, existem algumas restrições quanto ao comando SQL definido.

Por causa desses motivos, talvez seja então necessário trabalhar com a propriedade **Result Live** igual a “False” e utilizar um outro artifício para tornar o “result live” atualizável sem restrições no comando SQL e sem perder as linhas inseridas.

Utilizando Cached Updates

A outra forma de tornar o “result set” de um componente *TQuery* atualizável é utilizando o recurso de “cached updates” do Delphi.

Quando a propriedade **CachedUpdates** está ligada, todas as inserções, alterações e exclusões realizadas sobre o *Dataset* não são enviadas diretamente para o banco. Ao invés

disso, são armazenadas em um “cache” local na memória até que se dispare um comando do *Dataset* para aplicar todas as atualizações através de único processo no banco de dados.

Com isso, toda vez que for realizado um comando **post**, as alterações serão enviadas para o “cache” ao invés de irem diretamente para o banco, seja através do *DBNavigator*, automaticamente quando se deixa a linha ou através da chamada explícita do comando.

Mesmo se quisermos enviar linha a linha os registros para o banco de dados, talvez seja necessária a utilização do recurso de “cached updates” simplesmente para tornar o “result set” atualizável, caso não seja possível utilizar a propriedade **Result Live** do *TQuery*.

Para utilizar o recurso de “cached updates”, deve-se ligar a propriedade de mesmo nome do componente *TQuery*. Vamos fazer isso para o *QProduto*, mas mantendo por enquanto a propriedade **Result Live** = “True”.

Vamos retirar do *DBNavigator* alguns botões deixando somente os botões de navegação. A seguir, vamos colocar quatro novos botões do tipo *SpeedButton* para implementarmos as operações de inserção, exclusão, salvamento e cancelamento.

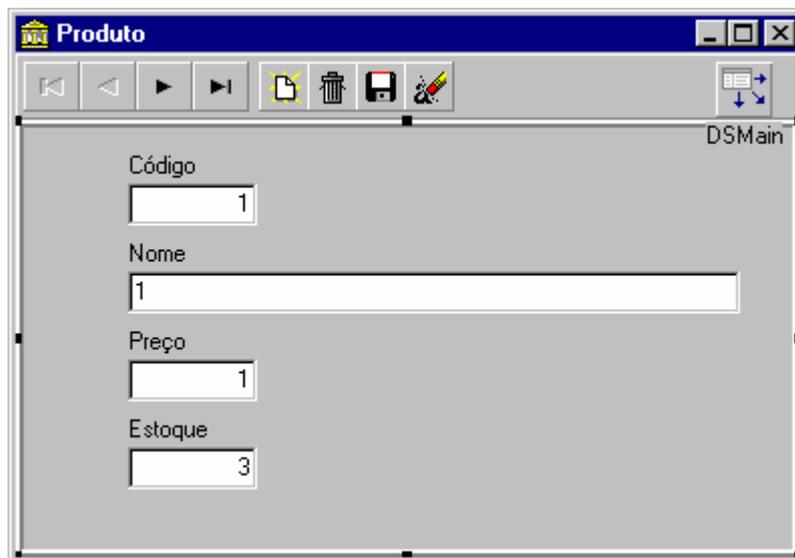


Fig 7.5: O form de Produto.

Componente	Propriedade	Valor
SppedButton1	Name	btnAppend
SppedButton2	Name	btnDelete
SppedButton3	Name	btnSave
SppedButton4	Name	btnDiscard

Tabela de Propriedades

Devemos agora, implementar o código para os eventos **OnClick** dos botões.

O primeiro botão “btnAppend” deve apenas criar uma nova linha no “result set” através do método **append** do componente *QProduto*. Entretanto vamos tentar ser um pouco mais genérico no código para depois podermos reutilizar o código escrito através de “templates”.

```
procedure TfrmProduto.btnAppendClick(Sender: TObject);
begin
    DSMain.DataSet.Append;
```

```
end;
```

Para o segundo botão, também não há nenhuma novidade.

```
procedure TfrmProduto.btnDeleteClick(Sender: TObject);  
begin  
    DSMain.DataSet.Delete;  
end;
```

Usaremos o terceiro botão “btnSave” para aplicar no banco de dados todas as alterações feitas nos diversos registros e que estão por enquanto armazenadas no “cached updates”.

```
procedure TfrmProduto.btnSaveClick(Sender: TObject);  
begin  
    TBDEDataSet(DSMain.DataSet).ApplyUpdates;  
    TBDEDataSet(DSMain.DataSet).CommitUpdates;  
end;
```

Nesse caso, foi necessário fazer um “typecast” na propriedade **DataSet** para que ela possa identificar os métodos **ApplyUpdates** e **CommitUpdates**. Esses dois métodos são responsáveis em aplicar as alterações pendentes no “cache” no banco de dados.

Além disso, é necessário utilizar a “unit” que contém a classe *TBDEDataSet*. Podemos fazer isso na cláusula **use** da seção **implementation**.

- **ApplyUpdates**: esse método aplica no banco de dados todas as alterações pendentes no “cached update”.
- **CommitUpdates**: esse método limpa do “buffer” do “cached update” os registros que foram aplicados no banco, após uma atualização realizada com sucesso.

Para o último botão (“btnDiscard”), podemos utilizar o método **CancelUpdates** que também limpa o “buffer”, mas descartando todas as alterações atualmente contidas no “cached update”.

```
procedure TfrmProduto.btnDiscardClick(Sender: TObject);  
begin  
    TBDEDataSet(DSMain.DataSet).CancelUpdates;  
end;
```

Desta forma, temos a tela de produtos funcionando através do recurso “cached updates”, ou seja, as alterações sendo enviadas em conjunto para o banco de dados.

Além disso, essa é uma alternativa para criar “result sets” atualizáveis sem utilizar a propriedade **Request Live**, como veremos a seguir.

Utilizando o Componente TUpdateSQL

Existem duas maneiras de se tornar o “result set” de uma “query” atualizável, sem as limitações impostas pela definição da propriedade **Request Live**. A primeira é utilizando o

componente *TUpdateSQL*. Através desse componente é possível definir os comandos que serão utilizados para efetuar as operações de inserção, alteração e exclusão. A segunda forma é através do evento **OnUpdateRecord** e será vista mais adiante nos próximos capítulos.

Vamos então colocar um componente *TUpdateSQL* no *DataModule* da aplicação, como mostrado na figura, e definir algumas propriedades através da tabela a seguir.



Fig 7.6: Componente *USProduto*.

Componente	Propriedade	Valor
UpdateSQL1	Name	USProduto
QProduto	Request Live	False
	UpdateObject	USProduto

Tabela de Propriedades

Pressionando o botão direito do mouse sobre o componente, pode-se seleccionar a opção **Update SQL Editor...** para abrir uma tela que irá permitir a geração dos comandos SQLs de atualização simplesmente definindo alguns parâmetros.

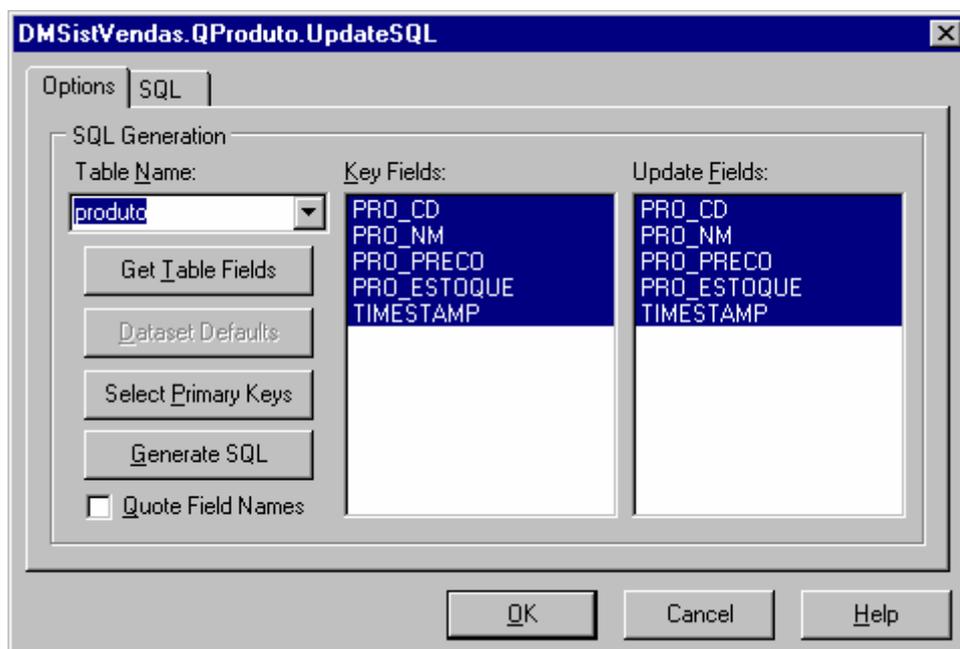


Fig 7.7: Update SQL Editor.

Na lista **Table Name** irão aparecer as tabelas que fazem parte da cláusula **From** do comando SQL. Deve-se escolher na lista a tabela que irá sofrer as atualizações. Após selecionada a tabela, pode-se pressionar o botão **Get Table Fields** para trazer os campos

dessa tabela para as listas à direita da tela. Se existir somente uma tabela, toda esse procedimento não será necessário, já que a tabela e seus campos serão trazidos automaticamente.

As duas listas da direita apresentam todos os campos contidos na tabela. Deve-se selecionar na primeira lista “Key Fields”, os campos que serão utilizados na cláusula **where** dos comandos **update** e **delete**. Pode-se optar somente pela chave, caso se queira deixar a concorrência mais livre, ou escolher todos os campos para verificar se o registro já foi alterado por outro usuário. Desta forma, consegue-se simular os valores “upWhereKeyOnly” e “upWhereAll” respectivamente da propriedade **UpdateMode** do *DataSet*, já que essa propriedade só é válida quando se utiliza a propriedade **Request Live** igual a “True”.

Para selecionar apenas a chave primária, pode-se utilizar o botão **Select Primary Keys**. Em nosso exemplo, vamos utilizar esse botão para selecionar apenas a chave primária.

Deve-se selecionar na lista **UpdateFields**, os campos que serão alterados e inseridos pelos comandos **update** e **insert**.

Finalizando, o último botão serve para gerar os comandos SQLs, seguindo o que já foi definido.

Depois de gerado, os comandos podem ainda sofrer modificações. Pode-se por exemplo retirar a coluna PRO_CD da cláusula **set** do comando **update**, já que não devemos deixar o usuário alterar a chave primária. Isso pode até causar um erro em alguns bancos de dados.

Pode-se notar que em alguns lugares foram utilizados prefixos “OLD” antes do nome das colunas. Isso foi feito para que seja testado o valor anterior a modificação realizada ao invés do valor atual. Esse processo é necessário quando utiliza-se todos os campos na cláusula **where**, para permitir que os campos sejam alterados e a verificação seja feita a partir de seus valores anteriores.

Gravação Linha a Linha ou em Batch

No início do exemplo, as gravações eram feitas linha a linha. Ao sair de um registro para o outro, as alterações eram automaticamente gravadas no banco de dados. Entretanto, este comportamento foi alterado quando se optou em trabalhar com “cached update”. Apesar do recurso “cached update” apresentar tais características, não foi por esse motivo que nós o utilizamos em nosso exemplo. O motivo principal de termos utilizado o recurso de “cached updates” foi para permitir que um comando qualquer SQL, sem limitações, pudesse gerar um “result set” atualizável.

Qual será então a melhor forma de trabalhar? Fazer as gravações linha a linha ou armazená-las em um “cache” e depois enviá-las todas de uma vez para o servidor. Cada uma das alternativas possuem vantagens e desvantagens que devem ser observadas antes de se escolher o processo que será utilizado.

Pode-se citar como vantagens de se trabalhar com atualizações em “batch”, ou seja, armazenando no “cache” e enviando o conjunto de atualizações de uma única vez para o banco:

- Os pacotes da rede serão melhor dimensionados, além de diminuir o número de comandos que serão enviados pela rede, tendo no total um tráfego bem menor na rede.
- Se um controle de transação estiver sendo utilizado, permite que a atualização seja uma operação única. Se um dos comandos falhar, nenhum comando é efetivado e o banco de dados volta para o estado original antes do início da transação.

Porém este método também possui algumas desvantagens:

- Esse método pode confundir o usuário. O usuário pode perder o controle do que ele já atualizou. Além disso, se um erro ocorrer na gravação, a correção do registro ou dos registros pelo usuário é um processo difícil de ser implementado.
- Como o usuário pode levar muito tempo para alterar os registros, existe uma probabilidade maior de um outro usuário ter alterado o mesmo registro e portanto bloquear a gravação.
- Outro fator importante a ser considerado é o tempo de espera do usuário. Em um sistema, o usuário nunca gosta de esperar muito tempo por um determinado processo. A gravação linha a linha distribui o processo total de atualização em pequenos intervalos de tempo que são muitas vezes consumidos pelo próprio tempo de digitação do usuário tornando-os imperceptíveis. Já a gravação em “batch”, dependendo da complexidade do processo que está sendo realizado, pode demorar muito, já que toda a atualização é feita de uma única vez. O tempo total da atualização em “batch” é até menor do que os feitos linha a linha, já que um número menor de comandos são enviados ao banco. Entretanto o usuário poderá achar a performance da gravação linha a linha bem melhor do que a outra, porque o tempo total é diluído em diversas operações de tempo bem menores.

Através da discussão acima, pode-se concluir que as gravações linha a linha parecem ser um método mais interessante de se utilizar, a menos que haja a necessidade da gravação ser realizada dentro de uma única transação, como por exemplo, o cadastro de um pedido e de seus itens.

Para fazer isso no Delphi, podemos continuar utilizando o recurso “cached updates”, mas questionando ao usuário o salvando ou cancelamento das alterações antes que ele deixe o registro.

Antes de fazer isso, vamos organizar um pouco mais nossa aplicação. Ao invés de implementarmos a lógica diretamente nos eventos dos botões, vamos criar **procedures** separadas para cada evento para que estes possam depois serem reutilizados mais facilmente.

```
.  
.   
public  
  { Public declarations }  
  procedure Save;  
  procedure NewForInsert;  
  procedure Delete;  
  procedure Discard;  
end;  
.   
.
```

Depois de criar as procedures, vamos transferir o código contido nos eventos dos botões para as novas procedures:

```
procedure TfrmProduto.btnAppendClick(Sender: TObject);
begin
  NewForInsert;
end;

procedure TfrmProduto.btnDeleteClick(Sender: TObject);
begin
  Delete;
end;

procedure TfrmProduto.btnSaveClick(Sender: TObject);
begin
  Save;
end;

procedure TfrmProduto.btnDiscardClick(Sender: TObject);
begin
  Discard;
end;

procedure TfrmProduto.NewForInsert;
begin
  DSMain.DataSet.Append;
end;

procedure TfrmProduto.Delete;
begin
  DSMain.DataSet.Delete;
end;

procedure TfrmProduto.Save;
begin
  TBDEDataSet(DSMain.DataSet).ApplyUpdates;
  TBDEDataSet(DSMain.DataSet).CommitUpdates;
end;

procedure TfrmProduto.Discard;
begin
  TBDEDataSet(DSMain.DataSet).CancelUpdates;
end;
```

Para facilitar o gerenciamento dos eventos é interessante criar uma variável que indique se um determinado evento está sendo executado. Através desse método, pode-se verificar dentro de evento, se ele está sendo executado por um outro evento e assim bloquear chamadas múltiplas de um mesmo evento.

Vamos criar o seguinte tipo de dado na seção **type** da **interface**:

```
TOperState=(opNone,opNewForInsert,opDelete,opSave,opDiscard);
```

Na seção **public** da classe **TfrmProduto**, podemos criar uma variável desse tipo:

```
.
.
public
{ Public declarations }
OperState: TOperState;
procedure Save;
procedure NewForInsert;
```

```
    procedure Delete;  
    procedure Discard;  
end;  
.
```

Podemos inicializá-la no evento **OnCreate** da *Form*.

```
procedure TfrmProduto.FormCreate(Sender: TObject);  
begin  
    OperState:=opNone;  
end;
```

Agora, para cada evento devemos controlar o valor da variável atribuindo a ela a operação que está sendo executada e retornando ao valor original no final do evento.

```
procedure TfrmProduto.NewForInsert;  
var OldOperState: TOperState;  
begin  
    OldOperState:=OperState;  
    OperState:=opNewForInsert;  
    try  
        DSMain.DataSet.Append;  
    finally  
        OperState:=OldOperState;  
    end;  
end;
```

Com isso, já temos o código mais organizado e com mais poder de gerenciamento. Vamos agora implementar o salvamento linha a linha. Para isso vamos utilizar o evento **OnUpdateData** do *DataSource* que é disparado sempre que se tentar enviar uma linha alterada para o “cache”, ou seja, sempre que é executado um comando **post**.

Nesse evento iremos perguntar ao usuário se ele deseja gravar o registro, descartar as alterações ou cancelar a tentativa de sair do registro.

```
procedure TfrmProduto.DSMainUpdateData(Sender: TObject);  
var ret: integer;  
begin  
    If OperState in [opNone,opNewForInsert] then begin  
        ret:=Application.MessageBox('Deseja salvar as alterações',  
            'Confirmação', MB_YESNOCANCEL + MB_ICONQUESTION );  
        case ret of  
            idYes: Save;  
            idNo: Discard;  
            idCancel: Abort;  
        end;  
    end;  
end;
```

A primeira coisa que será feita no evento é checar se nenhuma operação conhecida está sendo executada. Não queremos, por exemplo, que a pergunta seja feita se o usuário apertar o botão de salvar.

Depois será feita a pergunta para o usuário e com a resposta foi montado um comando **case**.

Se o usuário quiser salvar, simplesmente chamamos o comando **Save**. Se ele não quiser, descartamos a alteração antes de sair do registro. E se ele quiser cancelar, executamos o comando **Abort**.

Podemos melhorar o evento **Discard** para descartar e limpar o “cache” apenas se ele conter algum registro. Caso contrário podemos apenas descartar as alterações antes mesmo delas irem para o “cache”.

```
procedure TfrmProduto.Discard;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opDiscard;
  try
    If TBDEDataSet(DSMain.DataSet).UpdatesPending then
      TBDEDataSet(DSMain.DataSet).CancelUpdates
    else
      TBDEDataSet(DSMain.DataSet).Cancel;
  finally
    OperState:=OldOperState;
  end;
end;
```

O método **cancel** do *DataSet* descarta as alterações que ainda não foram para o “cached updates”. Através da propriedade **UpdatesPending**, pode-se verificar se existem registros pendentes no “cache” que ainda não foram enviados para o banco.

Para finalizar, devemos acrescentar os comandos que efetivam o “cached updates” no banco após o exclusão de um registro no evento **Delete**.

```
procedure TfrmProduto.Delete;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opDelete;
  try
    DSMain.DataSet.Delete;
    TBDEDataSet(DSMain.DataSet).ApplyUpdates;
    TBDEDataSet(DSMain.DataSet).CommitUpdates;
  finally
    OperState:=OldOperState;
  end;
end;
```

Trabalhando com o TTable

Como foi visto, fazer uma simples tela de cadastro com o componente *TQuery* foi um pouco trabalhoso. O mesmo não acontece com o componente *TTable*, que poderia fornecer a mesma funcionalidade através de um processo muito mais simples.

Vamos implementar então uma outra tela de cadastro, mas agora utilizando o componente *TTable*. Para isso vamos colocar um componente *TTable* no *DataModule* *DMSistVendas*.

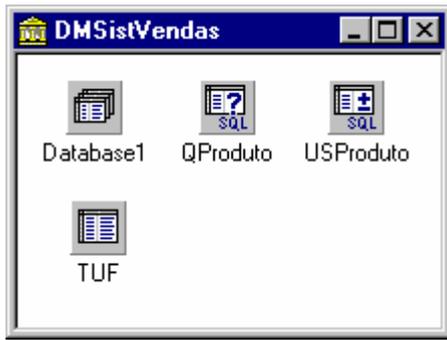


Fig 7.8: Componente *TTable*.

Componente	Propriedade	Valor
Table1	Name	TUF
	DatabaseName	DBVENDAS
	TableName	UF
	Active	TRUE

Tabela de Propriedades.

Utilizaremos este componente *TTable* para acessar a tabela de UF no banco de dados.

Vamos montar a tela de maneira parecida com a que montamos para a tela de Produto.

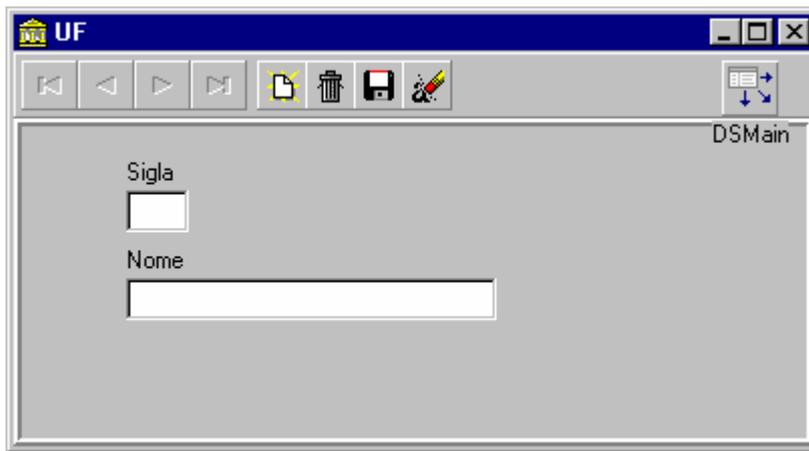


Fig 7.9: Tela de UF.

Componente	Propriedade	Valor
form1	Name	frmUF
	Caption	UF
Panel1	Caption	
Panel2	Caption	
DataSource1	Name	DSMain
	DataSet	DMSistVendas.TUF
DBNavigator	DataSource	DSMain
SppedButton1	Name	btnAppend
SppedButton2	Name	btnDelete
SppedButton3	Name	btnSave
SppedButton4	Name	btnDiscard
DBEdit1..DBEdit2	DataSource	DSMain
	DataField	UF_SG..UF_NM

Tabela de Propriedades.

Podemos implementar os eventos **OnClick** dos botões simplesmente executando os respectivos métodos do *DataSet* associado ao *DataSource DSMain*.

```
procedure TfrmUF.btnAppendClick(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).Append;
end;

procedure TfrmUF.btnDeleteClick(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).Delete;
end;

procedure TfrmUF.btnSaveClick(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).Post;
end;

procedure TfrmUF.btnDiscardClick(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).Cancel;
end;
```

Podemos agora alterar a tela principal do projeto para frmUF e executar a aplicação. Iremos notar que a tela funciona de uma forma bem semelhante à tela de Produto. Porque então implementar telas com *TQuery*, se através do componente *TTable* é muito mais simples e rápida a construção?

Além da facilidade, a implementação através do componente *TTable* é mais eficiente quando a transação é finalizada já que não busca todos os registros no banco. Entretanto, teremos que optar pela implementação através do *TQuery* quando existir um número maior de tabelas envolvidas em uma única tela. Cada *TTable* irá executar um comando **select**, que poderia talvez, ser executado por um único comando através do componente *TQuery* com uma performance bem melhor. Outra pequena desvantagem do componente *TTable* é que ele necessita buscar as informações de catálogo o que faz demorar mais na primeira abertura de cada tabela. Porém com a opção **ENABLED SCHEMA CACHE** do BDE ligada, esse problema pode ser minimizado.

Filtrando Registros

Esse capítulo mostra algumas possibilidades de seleção que podem ser apresentadas ao usuário

Em aplicações Cliente/Servidor é muito comum haver tabelas com milhares e até milhões de registros. Sendo assim, é preciso implementar consultas de forma a não trazer todos esses registros desnecessariamente para a aplicação. Em uma aplicação e principalmente em uma aplicação Cliente/Servidor onde a informação está localizada em um local físico diferente do aplicativo e esse local consegue prover uma inteligência capaz de manipular os dados, deve-se ter como regra trazer para aplicação somente as informações realmente necessárias para o usuário. Qualquer outra informação, além de poluir a tela, gera um tráfego adicional e desnecessário na rede. Muitas vezes tentamos adivinhar a informação que o usuário deseja ao invés de deixar que ele mesmo a peça. As melhores aplicações Cliente/Servidor são aquelas que conduzem os usuários através das informações realmente necessárias detalhando-as a medida do necessário.

As duas telas que nós construímos já começam ativas, ou seja, mostrando as primeiras linhas da tabela. Foi enviado um **select** para o banco de dados, algumas linhas foram trazidas para aplicação através da rede e nem sequer sabemos se o usuário as desejava ver. Imagine que a tabela de produto possua cerca de 1.000.000 de registros trazidos ordenadamente por nome. Na nossa tela de produto, o primeiro registro já iria aparecer quando o usuário abrisse a tela. Mas se o usuário desejasse ver um produto que começasse com 'M', será que ele iria querer navegar pelos botões de navegação até encontrá-lo?

Além disso, deve-se lembrar que quando se utiliza o componente *TQuery* para criar o “result set” e um comando **commit** é executado, todas as linhas restantes do “result set” são trazidas para aplicação. Imagine isso com 1.000.000 de registros.

Portanto, em aplicações Cliente/Servidor, principalmente em “result sets” que retornam uma grande quantidade de registro, é comum induzir o usuário a filtrar seu conjunto de registros antes de trazê-los para a aplicação. Existem várias formas de se implementar esse mecanismo na aplicação e a seguir vamos apresentar alguns deles.

QBE na mesma Tela de Manutenção

Uma forma de implementação é permitir o usuário realizar uma pesquisa através dos próprios campos utilizados para manutenção. Cria-se uma nova linha em branco e o usuário atribui valores para os campos que serão utilizados na pesquisa. Esse recurso é conhecido como QBE (Query by Exemplo), que é o processo de criar um comando SQL dinamicamente através de valores atribuídos aos campos da tabela. Somente os campos que possuem valores entram no filtro. Portanto o comando SQL **select** tem que ser montado dinamicamente durante a execução da aplicação. Esse recurso fornece uma boa flexibilidade nas consultas permitindo o usuário chegar bem próximo do dado antes de utilizar os recursos de navegação.

Podemos implementar esse recurso na tela de produto. Para isso vamos acrescentar dois botões para permitir a pesquisa. O primeiro para criar uma nova linha em branco que permita o usuário atribuir valores aos campos. O segundo para disparar a pesquisa.

Vamos também desligar a propriedade **Active** do *DataSet QProduto* para que inicialmente nenhum registro seja trazido.

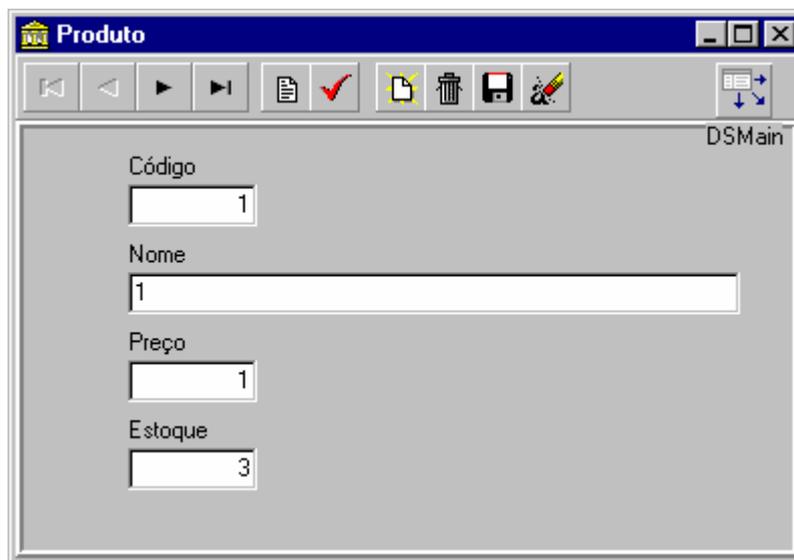


Fig 8.1: Tela de Produto.

Componente	Propriedade	Valor
SppedButton1	Name	btnNewForSearch
SppedButton2	Name	btnSearch

Tabela de Propriedades

Devemos então implementar o evento **OnClick** dos botões e as procedures da seguinte maneira:

```
procedure TfrmProduto.btnNewForSearchClick(Sender: TObject);
begin
    NewForSearch;
end;
```

```
procedure TfrmProduto.btnSearchClick(Sender: TObject);
begin
```

```
Search;
end;

procedure TfrmProduto.NewForSearch;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opNewForSearch;
  try
    If Not TBDEDataSet(DSMain.DataSet).Active then begin
      TQuery(DSMain.DataSet).SQL.Text:=CmdSelect + CmdSelectNull;
      TBDEDataSet(DSMain.DataSet).Open;
    end;
    TBDEDataSet(DSMain.DataSet).Append;
  finally
    OperState:=OldOperState;
  end;
end;

procedure TfrmProduto.SearchClick;
var OldOperState: TOperState;
    sQBE: String;
begin
  ActiveControl:=Nil;
  OldOperState:=OperState;
  OperState:=opSearch;
  try
    sQBE:=BuildQBE;
    TBDEDataSet(DSMain.DataSet).close;
    TQuery(DSMain.DataSet).SQL.Text:=CmdSelect+ ' where ' + sQBE + CmdOrderBy;
    TBDEDataSet(DSMain.DataSet).Open;
  finally
    OperState:=OldOperState;
  end;
end;
```

A procedure **NewForInsert** deve ser alterada para suportar a inclusão da linha se o *DataSet* estiver fechado.

```
procedure TfrmProduto.NewForInsert;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opNewForInsert;
  try
    If Not TBDEDataSet(DSMain.DataSet).Active then begin
      TQuery(DSMain.DataSet).SQL.Text:= CmdSelect + CmdSelectNull;
      TBDEDataSet(DSMain.DataSet).Open;
    end;
    DSMain.DataSet.Append;
  finally
    OperState:=OldOperState;
  end;
end;
```

Devemos cuidar de mais alguns detalhes antes de executar a aplicação:

Vamos acrescentar duas novas operações ao tipo **TOperState**:

```
TOperState=(opNone,opNewForInsert,opDelete,opSave,opDiscard,opNewForSearch,opSearch);
```

Devemos criar mais três variáveis na seção **private** do *form* para controlar o comando **select**:

```
.
.
private
  { Private declarations }
  CmdSelect: String;
  CmdOrderBy: String;
  CmdSelectNull: String;
public
  { Public declarations }
.
.
```

Vamos também alterar a propriedade **SQL** do *QProduto* tirando a cláusula **order by**:

Componente	Propriedade	Valor
QProduto	Active	False
	SQL	select * from Produto

Tabela de Propriedades

Vamos acrescentar a inicialização das variáveis no evento **OnCreate** do *Form* e retornar o valor da propriedade **SQL** do *QProduto* para o valor original no evento **OnDestroy**.

```
procedure TfrmProduto.FormCreate(Sender: TObject);
begin
  OperState:=opNone;
  CmdSelect:=TQuery(DSMain.DataSet).SQL.Text;
  CmdOrderBy:=' order by pro_cd';
  CmdSelectNull:=' where pro_cd is null ';
end;
```

```
procedure TfrmProduto.FormDestroy(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).close;
  TQuery(DSMain.DataSet).SQL.Text:=CmdSelect;
end;
```

Finalmente, temos que implementar a função **BuildQBE**. Futuramente, deve-se trabalhar melhor essa função para suportar campos “datas” e tratar os campos calculados e “lookups” do *DataSet*.

```
function TfrmProduto.BuildQBE: String;
var sep:string;
    j:integer;
begin
  Sep:='';
  For j:=0 to DSMain.DataSet.FieldCount-1 do
    If (DSMain.DataSet.Fields[j].AsString <> '') then begin
      If DSMain.DataSet.Fields[j].DataType = ftString then
        Result:= Format('%s %s (%s like '%s%s')', [Result, Sep,
          DSMain.DataSet.Fields[j].FieldName,
          DSMain.DataSet.Fields[j].AsString,'%'])
      else
        Result:= Format('%s %s (%s = %s)', [Result,Sep,
          DSMain.DataSet.Fields[j].FieldName,
          DSMain.DataSet.Fields[j].AsString]);
      Sep:='And';
    end;
```

```
end;  
end;
```

Antes ainda de executar, vamos ajustar a abertura dos *forms* no **Project/Options**. Deve-se alterar o **Main Form** novamente para 'frmProduto' e arrastar o 'DMSistVendas' para o topo da lista dos Auto-Create para que ele seja o primeiro a ser criado.

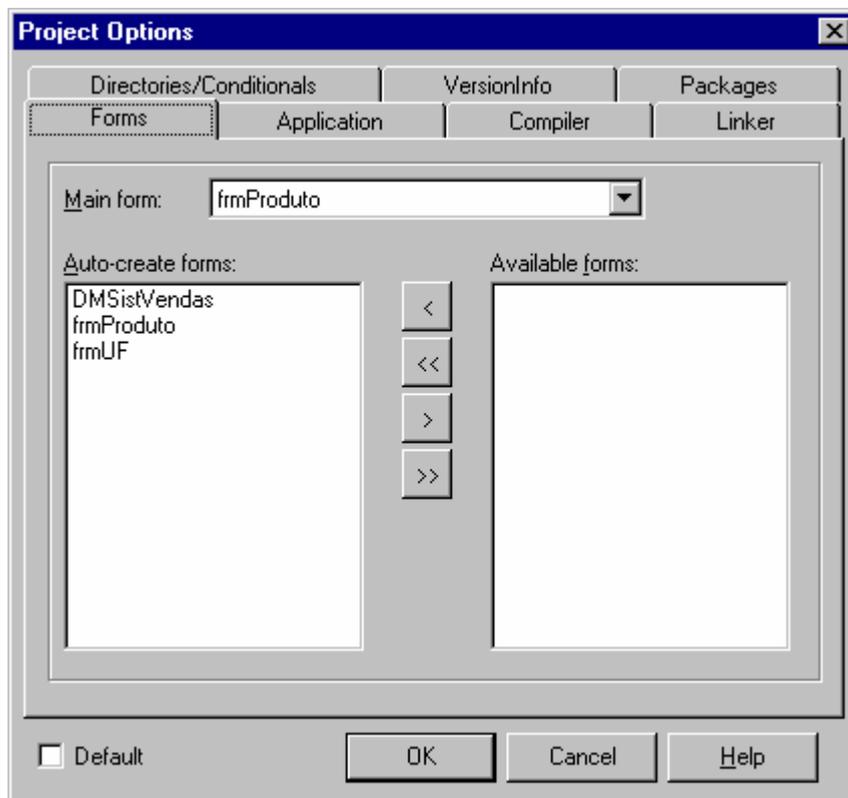


Fig 8.2: Project/Options.

Controlando os Estados da Tela

Para finalizar a tela de produtos, deve-se utilizar os estados da tela para controlar a habilitação dos botões de edição. O componente *Query* do Delphi já possui internamente uma máquina de estado controlada através do atributo **state**, onde os principais estados são: dsInactive, dsBrowse, dsEdit e dsInsert.

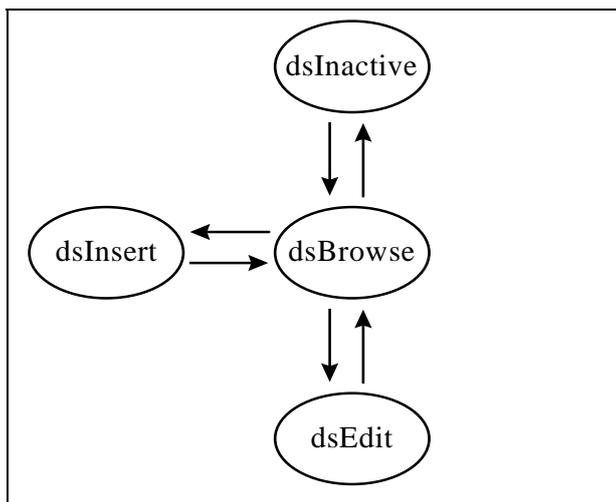


Fig 8.3: Estados do DataSet.

Entretanto esses estados não são suficientes para representar a tela de produto. É necessária a inclusão de um novo estado na máquina de estados, ficando da seguinte forma:

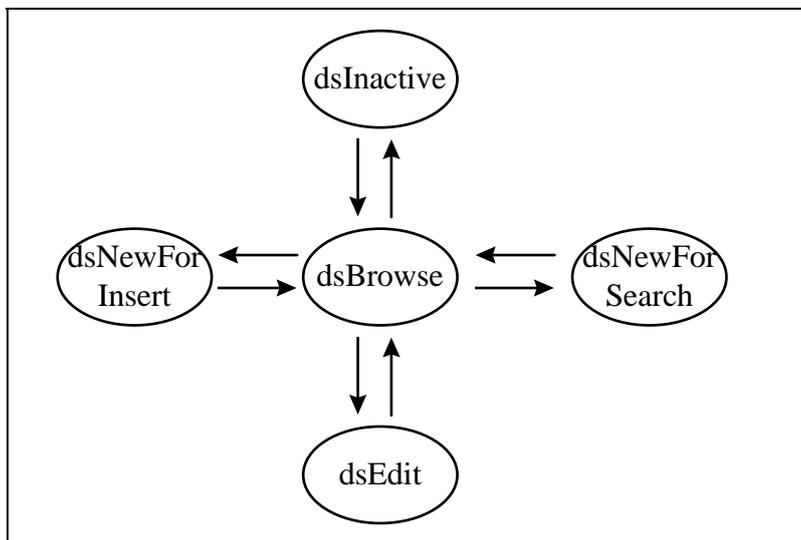


Fig 8.4: Nova máquina de estados.

Em nosso exemplo, podemos criar a máquina de estado no próprio *form* através da declaração de um tipo e uma variável.

```
TRecordState=(rdInactive,rdNewForSearch,rdNewForInsert,rdBrowse,rdEdit);
```

```
public
  { Public declarations }
  OperState: TOperState;
  RecordState: TRecordState;
  function BuildQBE: String;
  procedure Save;
  procedure NewForInsert;
  procedure Delete;
  procedure Discard;
  procedure NewForSearch;
  procedure Search;
```

```
procedure SetRecordState(Value: TRecordState);  
end;
```

Além disso, uma série de modificações nos eventos devem ser feitas e o código completo e final da tela de produto é apresentado abaixo. Pode-se facilmente transformar a tela de Produtos em um “Template” e adicionar no repositório de objetos do Delphi. Pode-se fazer como exercício, a tela de cidades utilizando-se o “template” construído. As únicas dependências da tela com a tabela de PRODUTO são as variáveis: **CmdOrderBy** e **CmdSelectNull**. Essas variáveis precisam então ser redefinidas no evento **OnCreate** do *Form* que herdar do “template” original.

```
unit produto;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
Db, ExtCtrls, StdCtrls, Mask, DBCtrls, Buttons;
```

```
type
```

```
TOperState=(opNone,opNewForInsert,opDelete,opSave,opDiscard,opNewForSearch,opSe  
arch);
```

```
TRecordState=(rdInactive,rdNewForSearch,rdNewForInsert,rdBrowse,rdEdit);
```

```
TfrmProduto = class(TForm)
```

```
Panel1: TPanel;
```

```
Panel2: TPanel;
```

```
DSMain: TDataSource;
```

```
DBNavigator1: TDBNavigator;
```

```
Label1: TLabel;
```

```
DBEdit1: TDBEdit;
```

```
Label2: TLabel;
```

```
DBEdit2: TDBEdit;
```

```
Label3: TLabel;
```

```
DBEdit3: TDBEdit;
```

```
Label4: TLabel;
```

```
DBEdit4: TDBEdit;
```

```
btnAppend: TSpeedButton;
```

```
btnDelete: TSpeedButton;
```

```
btnSave: TSpeedButton;
```

```
btnDiscard: TSpeedButton;
```

```
btnNewForSearch: TSpeedButton;
```

```
btnSearch: TSpeedButton;
```

```
procedure btnAppendClick(Sender: TObject);
```

```
procedure btnDeleteClick(Sender: TObject);
```

```
procedure btnSaveClick(Sender: TObject);
```

```
procedure btnDiscardClick(Sender: TObject);
```

```
procedure DSMainUpdateData(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
```

```
procedure btnNewForSearchClick(Sender: TObject);
```

```
procedure FormDestroy(Sender: TObject);
```

```
procedure btnSearchClick(Sender: TObject);
```

```
procedure DSMainStateChange(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
CmdSelect: String;
```

```
public
```

```
{ Public declarations }
```

```
CmdOrderBy: String;
```

```
CmdSelectNull: String;
```

```
OperState: TOperState;
```

```
RecordState: TRecordState;
```

```
function BuildQBE: String;
procedure Save;
procedure NewForInsert;
procedure Delete;
procedure Discard;
procedure NewForSearch;
procedure Search;
procedure SetRecordState(Value: TRecordState);
end;

var
  frmProduto: TfrmProduto;

implementation

uses Dm01, dbtables;

{$R *.DFM}

procedure TfrmProduto.FormCreate(Sender: TObject);
begin
  OperState:=opNone;
  SetRecordState(rdInactive);
  CmdSelect:=TQuery(DSMain.DataSet).SQL.Text;
  CmdOrderBy:=' order by pro_cd';
  CmdSelectNull:=' where pro_cd is null';
end;

procedure TfrmProduto.btnAppendClick(Sender: TObject);
begin
  NewForInsert;
end;

procedure TfrmProduto.btnDeleteClick(Sender: TObject);
begin
  Delete;
end;

procedure TfrmProduto.btnSaveClick(Sender: TObject);
begin
  Save;
end;

procedure TfrmProduto.btnDiscardClick(Sender: TObject);
begin
  Discard;
end;

procedure TfrmProduto.btnNewForSearchClick(Sender: TObject);
begin
  NewForSearch;
end;

procedure TfrmProduto.btnSearchClick(Sender: TObject);
begin
  Search;
end;

procedure TfrmProduto.NewForInsert;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opNewForInsert;
  try
    If Not TBDEDataSet(DSMain.DataSet).Active then begin
      TQuery(DSMain.DataSet).SQL.Text:= CmdSelect + CmdSelectNull;
      TBDEDataSet(DSMain.DataSet).Open;
    end;
  end;
end;
```

```
    end;
    DSMain.DataSet.Append;
    SetRecordState(rdNewForInsert);
  finally
    OperState:=OldOperState;
  end;
end;

procedure TfrmProduto.Delete;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opDelete;
  try
    DSMain.DataSet.Delete;
    TBDEDataSet(DSMain.DataSet).ApplyUpdates;
    TBDEDataSet(DSMain.DataSet).CommitUpdates;
    If TBDEDataSet(DSMain.DataSet).IsEmpty then begin
      TBDEDataSet(DSMain.DataSet).close;
      SetRecordState(rdInactive);
    end else
      SetRecordState(rdBrowse);
  finally
    OperState:=OldOperState;
  end;
end;

procedure TfrmProduto.Save;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opSave;
  try
    TBDEDataSet(DSMain.DataSet).ApplyUpdates;
    TBDEDataSet(DSMain.DataSet).CommitUpdates;
    SetRecordState(rdBrowse);
  finally
    OperState:=OldOperState;
  end;
end;

procedure TfrmProduto.Discard;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opDiscard;
  try
    If TBDEDataSet(DSMain.DataSet).UpdatesPending then
      TBDEDataSet(DSMain.DataSet).CancelUpdates
    else
      TBDEDataSet(DSMain.DataSet).Cancel;
    If TBDEDataSet(DSMain.DataSet).IsEmpty then begin
      TBDEDataSet(DSMain.DataSet).close;
      SetRecordState(rdInactive);
    end else
      SetRecordState(rdBrowse);
  finally
    OperState:=OldOperState;
  end;
end;

procedure TfrmProduto.NewForSearch;
var OldOperState: TOperState;
begin
  OldOperState:=OperState;
  OperState:=opNewForSearch;
  try
```

```
    If Not TBDEDataSet(DSMain.DataSet).Active then begin
        TQuery(DSMain.DataSet).SQL.Text:= CmdSelect + CmdSelectNull;
        TBDEDataSet(DSMain.DataSet).Open;
    end;
    TBDEDataSet(DSMain.DataSet).Append;
    SetRecordState(rdNewForSearch);
finally
    OperState:=OldOperState;
end;
end;

procedure TfrmProduto.FormDestroy(Sender: TObject);
begin
    TBDEDataSet(DSMain.DataSet).close;
    TQuery(DSMain.DataSet).SQL.Text:=CmdSelect;
end;

procedure TfrmProduto.Search;
var OldOperState: TOperState;
    sQBE: String;
begin
    ActiveControl:=Nil;
    OldOperState:=OperState;
    OperState:=opSearch;
    try
        sQBE:=BuildQBE;
        TBDEDataSet(DSMain.DataSet).close;
        TQuery(DSMain.DataSet).SQL.Text:= CmdSelect;
        If sQBE <> '' then
            TQuery(DSMain.DataSet).SQL.Text:=
                TQuery(DSMain.DataSet).SQL.Text + ' where ' + sQBE;
        TQuery(DSMain.DataSet).SQL.Text:=TQuery(DSMain.DataSet).SQL.Text +
            CmdOrderBy;
        TBDEDataSet(DSMain.DataSet).Open;
        If TBDEDataSet(DSMain.DataSet).IsEmpty then
            SetRecordState(rdNewForSearch)
        else
            SetRecordState(rdBrowse);
    finally
        OperState:=OldOperState;
    end;
end;

procedure TfrmProduto.DSMainUpdateData(Sender: TObject);
var ret: integer;
begin
    If OperState = opNone then begin
        If RecordState = rdNewForSearch then
            Discard
        else begin
            ret:=Application.MessageBox('Deseja salvar as alterações',
                'Confirmação', MB_YESNOCANCEL + MB_ICONQUESTION );
            case ret of
                idYes: Save;
                idNo: Discard;
                idCancel: Abort;
            end;
        end;
    end;
end;

procedure TfrmProduto.DSMainStateChange(Sender: TObject);
begin
    If DSMain.State=dsEdit then
        SetRecordState(rdEdit);
end;
```

```
function TfrmProduto.BuildQBE: String;
var sep:string;
    j:integer;
begin
    Sep:='';
    For j:=0 to DSMain.DataSet.FieldCount-1 do
        If (DSMain.DataSet.Fields[j].AsString <> '') then begin
            If DSMain.DataSet.Fields[j].DataType = ftString then
                Result:= Format('%s %s (%s like '%s%s')',
                    [Result,Sep,DSMain.DataSet.Fields[j].FieldName,
                    DSMain.DataSet.Fields[j].AsString,'%'])
            else
                Result:= Format('%s %s (%s = %s)',
                    [Result,Sep,DSMain.DataSet.Fields[j].FieldName,
                    DSMain.DataSet.Fields[j].AsString]);
            Sep:='And';
        end;
    end;
end;

procedure TfrmProduto.SetRecordState(Value: TRecordState);
begin
    RecordState:=Value;
    Case RecordState of
        rdInactive:
            begin
                btnNewForSearch.Enabled:=TRUE;
                btnAppend.Enabled:=TRUE;
                btnSearch.Enabled:=FALSE;
                btnSave.Enabled:=FALSE;
                btnDiscard.Enabled:=FALSE;
                btnDelete.Enabled:=FALSE;
            end;
        rdNewForSearch:
            begin
                btnNewForSearch.Enabled:=FALSE;
                btnAppend.Enabled:=FALSE;
                btnSearch.Enabled:=TRUE;
                btnSave.Enabled:=FALSE;
                btnDiscard.Enabled:=TRUE;
                btnDelete.Enabled:=FALSE;
            end;
        rdNewForInsert:
            begin
                btnNewForSearch.Enabled:=FALSE;
                btnAppend.Enabled:=FALSE;
                btnSearch.Enabled:=FALSE;
                btnSave.Enabled:=TRUE;
                btnDiscard.Enabled:=TRUE;
                btnDelete.Enabled:=FALSE;
            end;
        rdBrowse:
            begin
                btnNewForSearch.Enabled:=TRUE;
                btnAppend.Enabled:=TRUE;
                btnSearch.Enabled:=FALSE;
                btnSave.Enabled:=FALSE;
                btnDiscard.Enabled:=FALSE;
                btnDelete.Enabled:=TRUE;
            end;
        rdEdit:
            begin
                btnNewForSearch.Enabled:=FALSE;
                btnAppend.Enabled:=FALSE;
                btnSearch.Enabled:=FALSE;
                btnSave.Enabled:=TRUE;
                btnDiscard.Enabled:=TRUE;
                btnDelete.Enabled:=FALSE;
            end;
    end;
end;
```

```

        end;
    end;
end;

end.

```

Tela de Consulta Específica

Uma outra forma de filtrar a pesquisa que será feita no banco de dados para buscar o registro que o usuário deseja, é criar uma tela de consulta separada da tela de manutenção. Com isso a tela de manutenção não precisa conter elementos de navegação, já que pode trabalhar com apenas um registro de cada vez. O usuário utiliza a tela de consulta para filtrar, pesquisar e selecionar o registro desejado que é transferido para a tela de manutenção, onde normalmente informações mais detalhadas são apresentadas, permitindo ao usuário realizar as manutenções necessárias sobre esse registro.

Com esse tipo de padrão de cadastro é possível também, fazer um melhor balanceamento entre os dados necessários para consulta e os dados necessários para a manutenção, já que estes estão em telas diferentes. Para discutir esses conceitos, vamos implementar a tela de cliente na aplicação exemplo.

Tela de Consulta

Na tela de consulta explora-se a tabela ou as tabelas envolvidas, no sentido vertical, ou seja, um número maior de registros devem ser trazidos para aplicação para permitir que o usuário faça a escolha navegando entre eles. Entretanto, a seleção pode ser reduzida horizontalmente, já que a escolha do usuário pode ser feita exibindo-se um número pequeno de colunas, e conseqüentemente de tabelas, que o auxiliem na seleção. Para isso é preciso aplicar o seguinte conceito: informações de tabelas relacionadas à tabela principal não devem participar da seleção e sim dos filtros. Por exemplo, se o usuário necessitar escolher clientes de uma determinada cidade, ao invés de mostrar na consulta as cidades que o cliente pertence, faça-o informar antes da consulta a cidade da qual os clientes ele deseja. Através dessa técnica, elimina-se o número de tabelas necessárias na cláusula **from**, tornando o comando **select** bem mais eficiente com o mesmo resultado para o usuário que desejava selecionar um cliente de uma determinada cidade.

Iremos em nosso exemplo construir uma tela de consulta específica para selecionarmos o cliente. Antes porém, iremos criar alguns componentes *DataSets* no *DataModule*.



Fig 8.5: DataModule DMSistVendas

Componente	Propriedade	Valor
Query1	Name	QConsCliente

	DatabaseName	DBVENDAS
	SQL	SELECT PESSOA.PES_CD , PESSOA.PES_NM , PESSOA.PES_TP , PESSOA.PES_CGCCPF FROM CLIENTE CLIENTE , PESSOA PESSOA WHERE (CLIENTE.PES_CD = PESSOA.PES_CD) ORDER BY PESSOA.PES_NM
Table1	Name	TConsCliUF
	DatabaseName	DBVENDAS
	TableName	UF
Query2	Name	QConsCliCidade
	DatabaseName	DBVENDAS
	SQL	SELECT * FROM CIDADE WHERE UF_SG = :UF_SG
	Params	UF_SG String

Tabela de Propriedades

A figura a seguir mostra a tela de consulta de clientes. Note que são colocados na parte de cima da tela componentes *TEdit* para que o usuário possa fazer a pesquisa antes de trazer os dados para tela. Nessa seleção, encontram-se os campos UF e CIDADE que representam o relacionamento da tabela principal PESSOA com as tabelas UF e CIDADE. Portanto estas tabelas serão utilizadas para permitir o usuário fazer o filtro, mas não entram no **select** principal da tela.

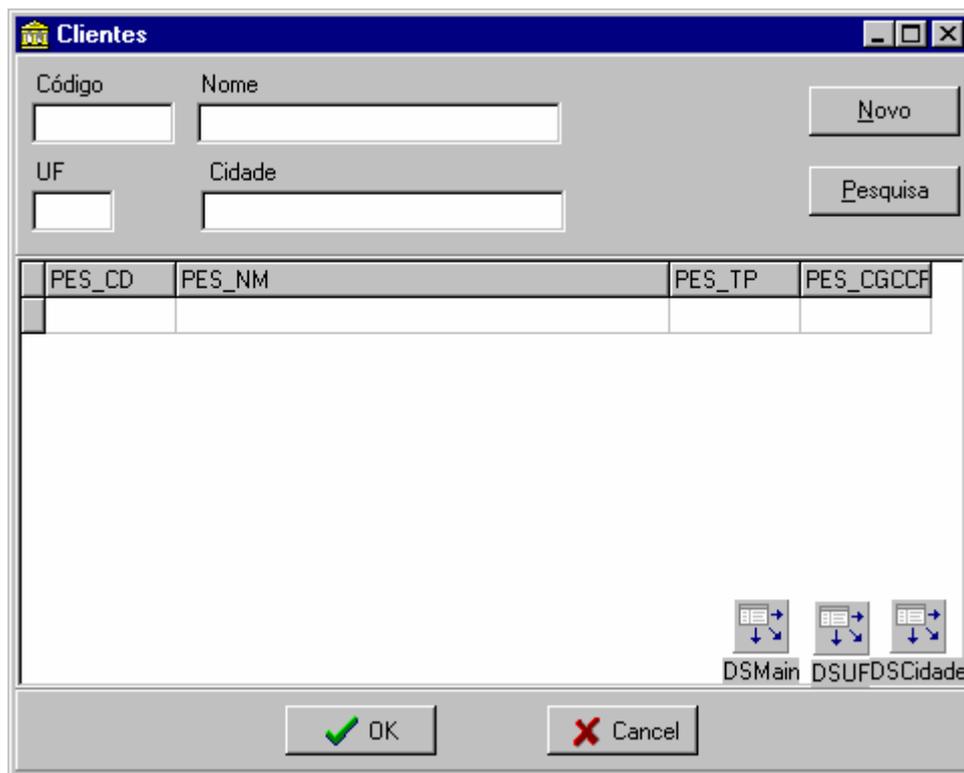


Fig 8.6: Tela de Consulta de Clientes.

Componente	Propriedade	Valor
form1	Name	frmConsCliente
	Caption	Clientes
Panel1	Align	alTop
Label1	Caption	Código
Label2	Caption	Nome

Label3	Caption	UF
Label4	Caption	Cidade
Edit1	Name	edtCodigo
Edit2	Name	edtNome
Edit3	Name	edtUF
Edit4	Name	edtCidade
Button1	Name	btnPesquisa
	Caption	Pesquisa
button2	Name	btnNovo
	Caption	Novo
DataSource1	Name	DSMain
	DataSet	DMSistVendas.QConsCliente
DataSource2	Name	DSUF
	DataSet	DMSistVendas.TConsCliUF
DataSource3	Name	DSCidade
	DataSet	DMSistVendas.QConsCliCidade
Panel2	Align	alClient
DbGrid1	DataSource	DSMain
Panel3	Align	alBottom
Bitbtn1	Kind	bkOk
Bitbtn2	Kind	bkCancel

Tabela de Propriedades

Para o usuário poder selecionar uma cidade e ou uma UF utilizaremos um Grid que será exibido quando ele posicionar no campo de UF ou de Cidade.

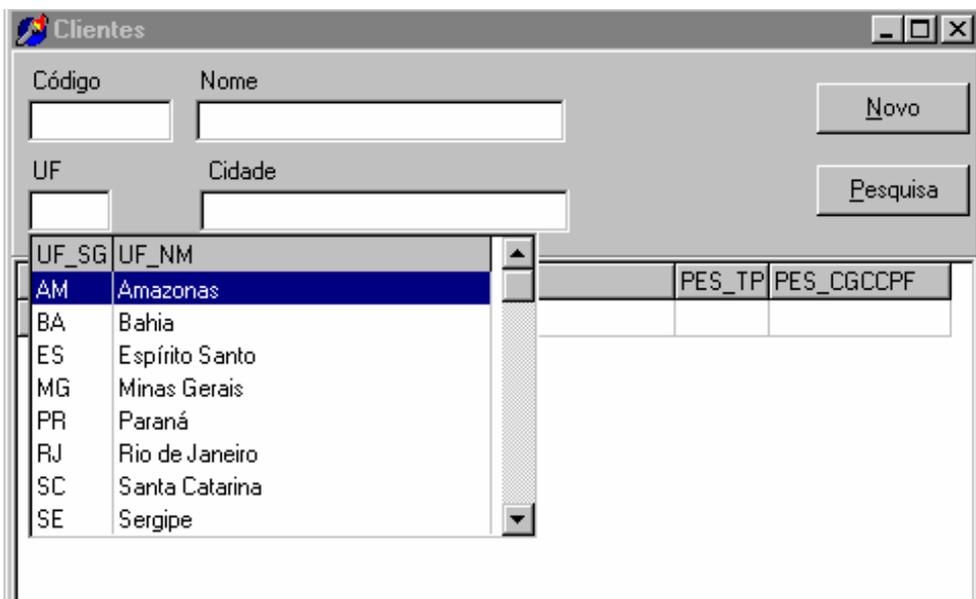


Fig 8.6: Grid para permitir a seleção de um UF pelo usuário.

Para fazermos essa consulta, iremos criar uma tela com um Grid dentro como mostrado na figura seguinte:

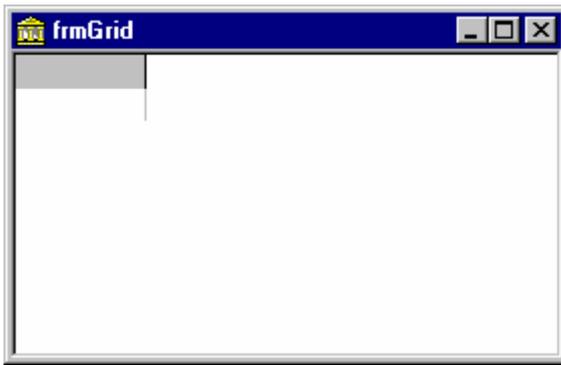


Fig 8.7: Grid para consulta.

Componente	Propriedade	Valor
Form1	Name	frmGrid
	BorderStyle	bsNone
DBGrid1	Align	alClient
	Options	[dgTitles, dgColumnResize, dgCollLines, dgTabs, dgRowSelect, dgConfirmDelete, dgCancelOnExit]

Tabela de Propriedades

Devemos implementar também os seguintes eventos:

```

procedure TfrmGrid.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  If Key = #13 then begin
    ModalResult:=mrOk;
  end else if key = #27 then begin
    ModalResult:=mrCancel;
  end;
end;

procedure TfrmGrid.FormClose(Sender:TObject; var Action:TCloseAction);
begin
  Action:=caFree;
end;

procedure TfrmGrid.DBGrid1DblClick(Sender: TObject);
begin
  ModalResult:=mrOk;
end;

procedure TfrmGrid.FormShow(Sender: TObject);
var p1:TPoint;
begin
  P1.x:=TForm(Owner).ActiveControl.Left;
  P1.y:=(TForm(Owner).ActiveControl.Top + TForm(Owner).ActiveControl.Height);
  P1:=TControl(TForm(Owner).ActiveControl.Parent).ClientToScreen(P1);
  If (P1.y + 150) > Screen.Height then
    P1.y:=P1.y - 150 - TForm(Owner).ActiveControl.Height;
  SetBounds(P1.x,P1.y,250,150);
end;

```

A implementação do evento **FormShow** faz com que a *Form* seja aberta exatamente embaixo dos componentes *edits* da tela de consulta.

O código da tela de consulta fica então da seguinte forma:

```

unit conscliente;

```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, Buttons, StdCtrls, Grids, DBGrids, ExtCtrls, DBCtrls;

type
  TfrmConsCliente = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    DBGrid1: TDBGrid;
    edtCodigo: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    edtNome: TEdit;
    btnPesquisa: TButton;
    Panel3: TPanel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    DSMain: TDataSource;
    btnNovo: TButton;
    DSUF: TDataSource;
    DSCidade: TDataSource;
    edtUF: TEdit;
    edtCidade: TEdit;
    Label3: TLabel;
    Label4: TLabel;
    procedure btnNovoClick(Sender: TObject);
    procedure btnPesquisaClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure edtUFEnter(Sender: TObject);
    procedure edtCidadeEnter(Sender: TObject);
  private
    { Private declarations }
    CmdSelect: String;
  public
    { Public declarations }
    sCodCidade: String;
  end;

var
  frmConsCliente: TfrmConsCliente;

implementation

uses Dm01, dbtables, ConsGrid;

{$R *.DFM}

procedure TfrmConsCliente.btnNovoClick(Sender: TObject);
begin
  edtCodigo.Clear;
  edtNome.Clear;
  edtUF.Clear;
  edtCidade.Clear;
end;

procedure TfrmConsCliente.btnPesquisaClick(Sender: TObject);
var sWhere, sSelect, sep: string;
    nPosOrderBy: integer;
begin
  TQuery(DSMain.DataSet).Close;

  Sep:='';
```

```
sWhere:=' And ' ;
sSelect:=CmdSelect;

If (edtCodigo.Text <> '') then begin
  sWhere:=Format('%s %s (%s = %s)',[sWhere,Sep,'PES_CD',edtCodigo.Text]);
  Sep:='And';
end;
If (edtNome.Text <> '') then begin
  sWhere:=Format('%s %s (%s like ''%s%s'') ',
                [sWhere,Sep,'PES_NM',edtNome.Text,'%']);
  Sep:='And';
end;
If (edtUF.Text <> '') then begin
  sWhere:=Format('%s %s (%s = ''%s'') ',[sWhere,Sep,'UF_SG',edtUF.Text]);
  Sep:='And';
end;
If (edtCidade.Text <> '') then begin
  sWhere:=Format('%s %s (%s = %s) ',[sWhere,Sep,'CID_CD',sCodCidade]);
  Sep:='And';
end;

If Sep <> '' then begin
  nPosOrderBy:=Pos('ORDER BY', UpperCase(sSelect));
  if nPosOrderBy = 0 then
    sSelect:=sSelect + sWhere
  else
    Insert(sWhere,sSelect,nPosOrderBy);
end;

TQuery(DSMain.DataSet).SQL.Text:=sSelect;

TQuery(DSMain.DataSet).Open;

end;

procedure TfrmConsCliente.FormCreate(Sender: TObject);
begin
  CmdSelect:=TQuery(DSMain.DataSet).SQL.TExt;
  DSUF.DataSet.Open;
end;

procedure TfrmConsCliente.FormDestroy(Sender: TObject);
begin
  DSUF.DataSet.Close;
  DSCidade.DataSet.close;
  TQuery(DSMain.DataSet).SQL.TExt:=CmdSelect;
end;

procedure TfrmConsCliente.edtUFEnter(Sender: TObject);
begin
  frmGrid:=TFrmGrid.Create(Self);
  frmGrid.DBGrid1.DataSource:=DSUF;
  If (frmGrid.ShowModal=mrOk) and (Not DSUF.DataSet.IsEmpty) then
  begin
    TEdit(Sender).Text:=DSUF.DataSet['UF_SG'];
    edtCidade.Clear;
  end;
end;

procedure TfrmConsCliente.edtCidadeEnter(Sender: TObject);
begin
  DSCidade.DataSet.Close;
  TQuery(DSCidade.DataSet).Params[0].value:=edtUF.Text;
  DSCidade.DataSet.Open;

  frmGrid:=TFrmGrid.Create(Self);
  frmGrid.DBGrid1.DataSource:=DSCidade;
```

```

If (frmGrid.ShowModal=mrOk) and (Not DSCidade.DataSet.IsEmpty) then
begin
  TEdit(Sender).Text:=DSCidade.DataSet['CID_NM'];
  sCodCidade:=DSCidade.DataSet.FieldByName('CID_CD').AsString;
end;
end;

end.

```

Tela de Manutenção

Como somente um registro é trazido na tela de manutenção, pode-se explorar mais o sentido horizontal, trazendo informações mais detalhadas. Porém, respeitando ainda a regra de não trazer informações ainda não requisitadas pelo usuário. Por exemplo, pode ser que a tela da manutenção apresente-se dividida em “fichas” ou “página”. Se a informação de uma página tem que acessar uma outra tabela que não seja a tabela principal, pode-se esperar o usuário selecionar a página para então buscar essas informações no banco de dados.

Para implementarmos a tela de manutenção de clientes, devemos primeiro criar o *DataSet* responsável pela busca dos dados no banco. Utilizaremos um componente *TQuery* com um comando SQL selecionando dados de quatro tabelas do banco de dados: PESSOA, CLIENTE, CIDADE, UF. Além disso será feito um filtro pelo código da pessoa para que seja selecionado apenas um registro no banco de dados. Desta forma reduziremos o número de linhas e aumentaremos a quantidade de informações sobre um determinado cliente.

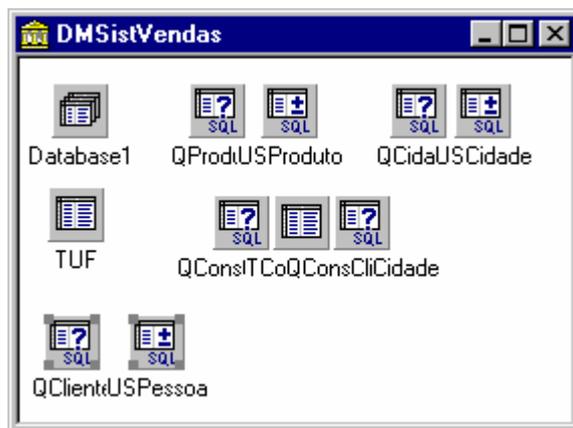


Fig 8.8: Data Module.

Componente	Propriedade	Valor
Query1	Name	Qcliente
	DatabaseName	DBVENDAS
	CachedUpdate	TRUE
	UpdateObject	USPessoa
	SQL	SELECT PESSOA.PES_CD, PESSOA.PES_NM, PESSOA.PES_TP, PESSOA.PES_CGCCPF, PESSOA.PES_LOGRADOURO, PESSOA.PES_NUMERO, PESSOA.PES_COMPLEMENTO, PESSOA.PES_BAIRRO, PESSOA.CID_CD, PESSOA.UF_SG, PESSOA.PES_DT_NASC, PESSOA.CID_CD_NASC, PESSOA.UF_SG_NASC, CLIENTE.CLI_LIMITECREDITO, CLIENTE.CLI_DEBITO, CIDADE.CID_NM,

		<pre> UF.UF_NM FROM PESSOA PESSOA, CLIENTE CLIENTE, CIDADE CIDADE, UF UF WHERE PESSOA.PES_CD = CLIENTE.PES_CD AND PESSOA.CID_CD = CIDADE.CID_CD AND PESSOA.UF_SG = CIDADE.UF_SG AND CIDADE.UF_SG = UF.UF_SG AND PESSOA.PES_CD= :PES_CD ORDER BY PESSOA.PES_NM </pre>
	Params	PES_CD Integer
UpdateSQL1	Name	USPessoa
	TableName	PESSOA

Tabela de Propriedades

Podemos agora, construir a tela de manutenção como mostra as figuras seguintes:

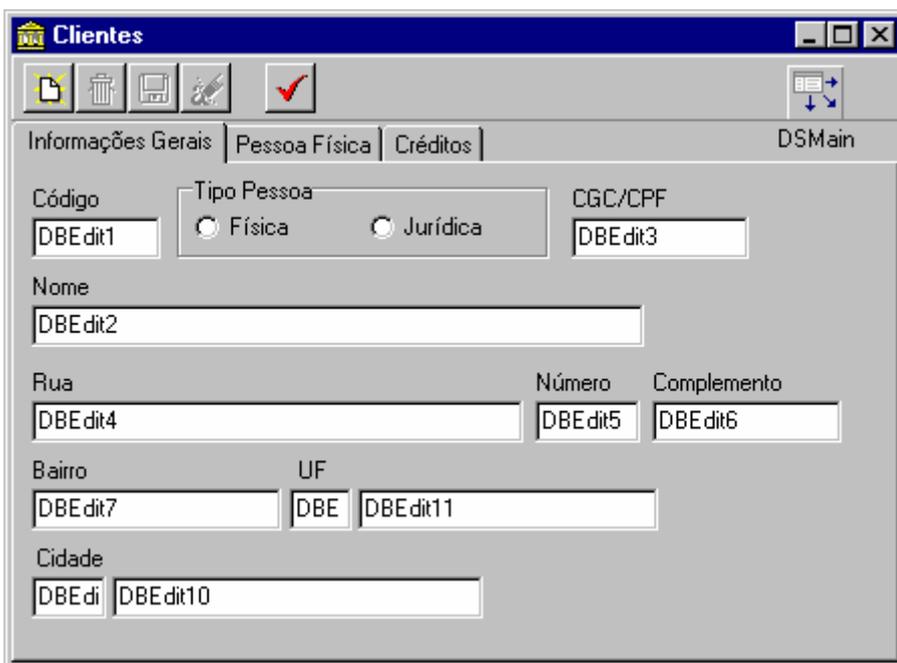


Fig 8.9: Tela de Manutenção de Clientes (Pág 1).

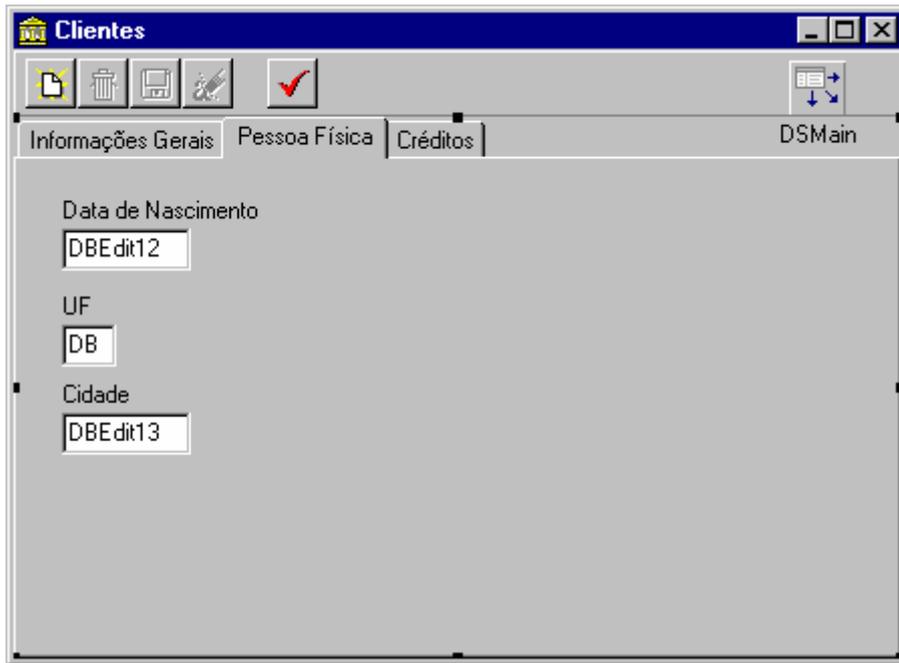


Fig 8.10: Tela de Manutenção de Clientes (Pág 2).

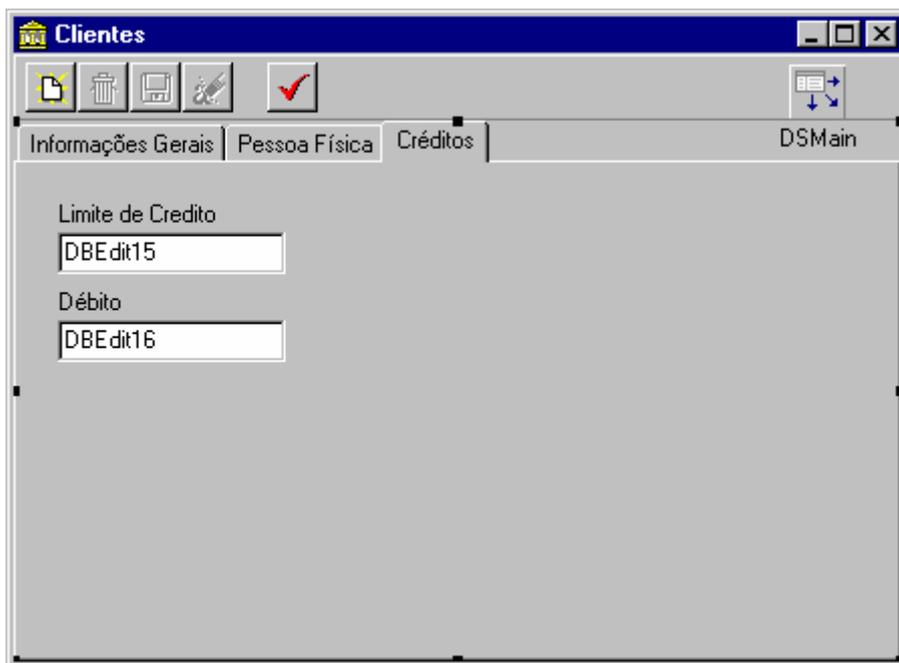


Fig 8.11: Tela de Manutenção de Clientes (Pág 3).

Componente	Propriedade	Valor
Form1	Name	frmCliente
	Caption	Clientes
Panel1	Align	alTop
SpeedButton1	Name	btnAppend
SpeedButton2	Name	btnDelete
SpeedButton3	Name	btnSave
SpeedButton4	Name	btnDiscard
SpeedButton5	Name	btnSearch

PageControl1	Align	alClient
TabSheet1	Caption	Informações Gerais
TabSheet2	Caption	Pessoa Física
TabSheet3	Caption	Créditos
DataSource1	Name	DSMain
	DataSet	DMSistVendas.Qcliente
DBControls

Tabela de Propriedades

Os eventos dos botões de edição serão definidos de maneira similar ao padrão utilizado pela tela de produto, ficando cada um da seguinte forma:

```

procedure TfrmCliente.btnAppendClick(Sender: TObject);
begin
  If Not DSMain.DataSet.Active then begin
    DSMain.DataSet.Open;
  end;
  DSMain.DataSet.Append;
end;

procedure TfrmCliente.btnDeleteClick(Sender: TObject);
begin
  DSMain.DataSet.Delete;
  TBDEDataSet(DSMain.DataSet).ApplyUpdates;
  TBDEDataSet(DSMain.DataSet).CommitUpdates;
  DSMain.DataSet.close;
end;

procedure TfrmCliente.btnSaveClick(Sender: TObject);
begin
  TBDEDataSet(DSMain.DataSet).ApplyUpdates;
  TBDEDataSet(DSMain.DataSet).CommitUpdates;
end;

procedure TfrmCliente.btnDiscardClick(Sender: TObject);
begin
  If TBDEDataSet(DSMain.DataSet).UpdatesPending then
    TBDEDataSet(DSMain.DataSet).CancelUpdates
  else
    TBDEDataSet(DSMain.DataSet).Cancel;
  If TBDEDataSet(DSMain.DataSet).IsEmpty then begin
    TBDEDataSet(DSMain.DataSet).close;
  end;
end;

```

O botão de “Pesquisa” irá chamar a tela de consulta de clientes para que o usuário possa escolher um determinado cliente para trabalhar. Depois de escolhido, informações mais detalhadas desse cliente são trazidas para a tela de manutenção para que possam ser visualizadas e editadas pelo usuário.

```

procedure TfrmCliente.btnSearchClick(Sender: TObject);
begin
  DSMain.DataSet.Close;
  frmConsCliente:=TfrmConsCliente.Create(Self);
  If (frmConsCliente.ShowModal=mrOk) and
    (Not (frmConsCliente.DSMain.DataSet['PES_CD'] = Null)) then
    begin
      TQuery(DSMain.DataSet).Params[0].value:=
        frmConsCliente.DSMain.DataSet['PES_CD'];
      DSMain.DataSet.Open;
    end;
end;

```

```
    frmConsCliente.free;  
end;
```

Podemos utilizar a própria máquina de estado do *DataSource* fornecida pelo Delphi para controlar a habilitação dos botões:

```
procedure TfrmCliente.DSMainStateChange(Sender: TObject);  
begin  
    Case DSMain.State of  
        dsInactive:  
            begin  
                btnAppend.Enabled:=TRUE;  
                btnDelete.Enabled:=FALSE;  
                btnSave.Enabled:=FALSE;  
                btnDiscard.Enabled:=FALSE;  
                btnSearch.Enabled:=TRUE;  
            end;  
        dsInsert:  
            begin  
                btnAppend.Enabled:=FALSE;  
                btnDelete.Enabled:=FALSE;  
                btnSave.Enabled:=TRUE;  
                btnDiscard.Enabled:=TRUE;  
                btnSearch.Enabled:=FALSE;  
            end;  
        dsEdit:  
            begin  
                btnAppend.Enabled:=FALSE;  
                btnDelete.Enabled:=FALSE;  
                btnSave.Enabled:=TRUE;  
                btnDiscard.Enabled:=TRUE;  
                btnSearch.Enabled:=FALSE;  
            end;  
        dsBrowse:  
            begin  
                btnAppend.Enabled:=TRUE;  
                btnDelete.Enabled:=TRUE;  
                btnSave.Enabled:=FALSE;  
                btnDiscard.Enabled:=FALSE;  
                btnSearch.Enabled:=TRUE;  
            end;  
    end;  
end;
```

Para finalizar devemos fechar o *DataSet* antes de sairmos da tela de manutenção, implementando o evento **OnClose** da seguinte forma:

```
procedure TfrmCliente.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    DSMain.DataSet.Close;  
end;
```

Recursos de LookUp

Existem várias formas de fornecer para o usuário uma pesquisa em registros de tabelas secundárias relacionadas com a tabela principal. Esse recurso, muitas vezes chamado de “lookup” deve prover implementações para as três fases seguintes:

- Preencher e exibir uma lista ou tabela de registros para permitir a escolha do usuário;

- Ao se escolher um registro, deve-se atualizar o código relacionado na tabela principal;
- Quando for selecionado um registro na tabela principal, deve-se selecionar o registro correspondente na tabela secundária, permitindo a exibição de alguns campos como a descrição.

Campo LookUp do Delphi

O Delphi permite a criação de campos “lookup’s” dentro de um *dataset* para apresentar uma lista com os registros de uma outra tabela relacionada. Esse recurso implementa as três fases discutidas acima da seguinte forma:

- Fase 1. A lista é preenchida através de um *DataSet* que seleciona os registros da tabela relacionada. É utilizado componentes *combo box* ou *list box* para apresentar a lista. Quando utilizado *combo box*, a parte *edit* não é habilitada para edição, forçando o usuário a realmente escolher um elemento válido da lista. Como é utilizado um componente *DataSet* para prover a seleção, é possível atribuir filtros ao “result set”, desde que não inviabilize a fase 3. Quando o recurso é implementado diretamente através dos componentes (*combo box* e *list box*) permite que o usuário visualize mais de um campo ao mesmo tempo para seleção. Para se abrir a lista de opções é necessário obviamente que o *DataSet* esteja aberto.
- Fase 2. Quando é selecionado um registro da lista, automaticamente é feita a atualização do campo da tabela principal relacionado com a chave primária da tabela utilizada para seleção.
- Fase 3. Não é necessária a participação da descrição ou de qualquer outro campo que se deseje visualizar da tabela secundária no comando **select** da tabela principal. Quando o campo (chave estrangeira) da tabela principal que se relaciona com a tabela secundária é trazido para a tela, automaticamente o Delphi procura no *DataSet* secundário o registro e mostra os campos descritivos da tabela secundária. Portanto, todas as linhas da tabela secundária devem estar disponíveis na estação cliente para que qualquer código seja encontrado. Por isso é preciso tomar bastante cuidado na atribuição de filtros para permitir que os registros necessários sejam encontrados pelo Delphi.

A principal vantagem desse método é a facilidade de implementá-lo no Delphi. Além disso ele simplifica o comando SQL utilizado para a tabela principal, já que não necessita da participação de outras tabelas no **select**.

Mas deve-se tomar cuidado com sua utilização quando as tabelas relacionadas possuem muitos registros porque basicamente todos os registros terão que ser trazidos para a máquina do usuário pela rede. Imagine selecionar através desse recurso uma tabela de clientes com cerca de 1.000.000 de registros. Apesar da possibilidade de se atribuir filtros ao *DataSet* responsável em popular a lista, isto é difícil devido a necessidade da fase três de encontrar o registro através do código. Se filtros forem colocados, pode ser que esse registro não seja encontrado.

Outro fator que deve ser observado é a quantidade de tabelas relacionadas com a tabela principal. Se a tabela principal possuir muitas tabelas relacionadas, para abrir a tabela principal é necessário abrir também todas as tabelas secundárias o que pode prejudicar muito a performance nesse ponto.

Outros Métodos

Apesar do nosso exemplo da tela de cliente tratar tabelas pequenas como UF e cidade, tentaremos mostrar a utilização de recursos alternativos quando o método “lookup” do Delphi não for satisfatório.

Fase 1. Utilizaremos o mesmo recurso utilizado na tela de consulta de clientes para propiciar a seleção de um registro secundário na tela de manutenção de clientes. Quando o usuário posicionar nos campos da tabela principal que se relacionam com as tabelas secundárias, um “grid” será exibido para permitir a escolha.

Fase 2. A atualização dos campos da tabela principal é feita quando o usuário fecha a tela de pesquisa escolhendo uma das opções.

Fase 3. Para evitar que os *DataSets* responsáveis pelas tabelas secundários necessitem ser abertos para a seleção de um registro da tabela principal, incluímos os campos de descrição no próprio comando **select**.

Mais adiante, nos próximos capítulos, tentaremos melhorar ainda mais esse recurso, permitindo ao usuário a atribuição de filtros.

Portanto os eventos **OnEnter** dos campos relacionados UF_SG e CID_CD ficam da seguinte forma:

```
procedure TfrmCliente.DBEdit9Enter(Sender: TObject);
begin
  If DSMain.State <> dsInactive then begin
    If Not DSUF.DataSet.Active then DSUF.DataSet.Open;
    frmGrid:=TFrmGrid.Create(Self);
    frmGrid.DBGrid1.DataSource:=DSUF;
    If (frmGrid.ShowModal=mrOk) and (Not DSUF.DataSet.IsEmpty) then begin
      If DSMain.State = dsBrowse then DSMain.DataSet.Edit;
      DSMain.DataSet['UF_SG']:=DSUF.DataSet['UF_SG'];
      DSMain.DataSet['UF_NM']:=DSUF.DataSet['UF_NM'];
      DSMain.DataSet['CID_NM']:= '';
      DSMain.DataSet['CID_CD']:=Null;
    end;
  end;
end;

procedure TfrmCliente.DBEdit8Enter(Sender: TObject);
begin
  If DSMain.State <> dsInactive then begin
    DSCidade.DataSet.Close;
    TQuery(DSCidade.DataSet).Params[0].AsString:=
      DSMain.DataSet.FieldByName('UF_SG').AsString;
    DSCidade.DataSet.Open;
    frmGrid:=TFrmGrid.Create(Self);
    frmGrid.DBGrid1.DataSource:=DSCidade;
    If (frmGrid.ShowModal=mrOk) and (Not DSCidade.DataSet.IsEmpty) then begin
      If DSMain.State = dsBrowse then DSMain.DataSet.Edit;
      DSMain.DataSet['CID_NM']:=DSCidade.DataSet['CID_NM'];
      DSMain.DataSet['CID_CD']:=DSCidade.DataSet['CID_CD'];
    end;
  end;
end;
```

Buscando Registros de outras TabSheets

Para não se “pesar” muito o comando **select** da tabela principal, não foi incluída a busca da descrição da UF e CIDADE de nascimento que estão localizados em uma outra *TabSheet*. Não vale a pena acrescentar mais duas tabelas na cláusula **from** do comando, sendo que o usuário pode nem se quer entrar na página onde se encontram os dados. Deve-se sempre seguir o lema de não trazer para a aplicação informações que o usuário ainda não pediu para ver ou editar. Se dados de uma tabela secundária só serão vistos se o usuário selecionar determinada *TabSheet*, pode-se postergar sua busca para quando a *TabSheet* for selecionada. Entretanto, quando as informações estão em uma mesma tabela já utilizada no comando **select** para trazer os dados da *TabSheet* inicial, não vale a pena enviar outro **select** para buscar o restante das informações de uma mesma tabela.

A implementação do código fica então como a seguir:

```

procedure TfrmCliente.PageControl1Change(Sender: TObject);
begin
  If (PageControl1.ActivePage = TabSheet2) and
    (DSMain.State = dsBrowse) then begin
    If DSMain.DataSet['UF_SG_NASC'] <> Null then begin
      DMSistVendas.QGeral.SQL.Text:=
        'select UF_NM from UF where UF_SG = ' +
          DSMain.DataSet['UF_SG_NASC'] + ''';
      DMSistVendas.QGeral.Open;
      EdtUFNmNasc.Text:=DMSistVendas.QGeral['UF_NM'];
      DMSistVendas.QGeral.close;
    end else begin
      EdtUFNmNasc.Text:='';
    end;
    If DSMain.DataSet['CID_CD_NASC'] <> Null then begin
      DMSistVendas.QGeral.SQL.Text:=
        'select CID_NM from Cidade where CID_CD = ' +
          DSMain.DataSet.FieldByName('CID_CD_NASC').AsString +
          ' and UF_SG = ' +
          DSMain.DataSet.FieldByName('UF_SG_NASC').AsString;
      DMSistVendas.QGeral.Open;
      EdtCidNmNasc.Text:=DMSistVendas.QGeral['CID_NM'];
      DMSistVendas.QGeral.close;
    end else begin
      EdtCidNmNasc.Text:='';
    end;
  end;
end;

procedure TfrmCliente.DBEdit14Enter(Sender: TObject);
begin
  If DSMain.State <> dsInactive then begin
    If Not DSUF.DataSet.Active then DSUF.DataSet.Open;
    frmGrid:=TFrmGrid.Create(Self);
    frmGrid.DBGrid1.DataSource:=DSUF;
    If (frmGrid.ShowModal=mrOk) and (Not DSUF.DataSet.IsEmpty) then begin
      If DSMain.State = dsBrowse then DSMain.DataSet.Edit;
      DSMain.DataSet['UF_SG_NASC']:=DSUF.DataSet['UF_SG'];
      DSMain.DataSet['CID_CD_NASC']:=Null;
      edtUFNmNasc.Text:=DSUF.DataSet['UF_NM'];
      edtCidNmNasc.Text:='';
    end;
  end;
end;

procedure TfrmCliente.DBEdit13Enter(Sender: TObject);
begin
  If DSMain.State <> dsInactive then begin

```

```

DSCidade.DataSet.Close;
TQuery(DSCidade.DataSet).Params[0].AsString:=
    DSMain.DataSet.FieldByName('UF_SG_NASC').AsString;
DSCidade.DataSet.Open;

frmGrid:=TFrmGrid.Create(Self);
frmGrid.DBGrid1.DataSource:=DSCidade;
If (frmGrid.ShowModal=mrOk)
    and (Not DSCidade.DataSet.IsEmpty) then begin
    if DSMain.State = dsBrowse then DSMain.DataSet.Edit;
    edtCidNmNasc.Text:=DSCidade.DataSet['CID_NM'];
    DSMain.DataSet['CID_CD_NASC']:=DSCidade.DataSet['CID_CD'];
    end;
end;
end;

```

Controlando Transações

Quando se trabalha com banco de dados relacionais, pode-se utilizar o recurso de transação para efetivar o processo como um todo ou voltar todas as atualizações desde o início do processo caso algum erro ocorra.

Em nosso exemplo, por enquanto, a manutenção de clientes está sendo feita somente na tabela de PESSOA. Para que ele fique correto de acordo com a lógica de negócio da aplicação, devemos atualizar também a tabela de CLIENTE nas manutenções feitas pelo usuário. Portanto, quando inserimos um registro devemos inseri-lo na tabela de PESSOA e também na tabela de CLIENTE. E essa inserção deve estar dentro de uma transação, para que esse processo seja feito como um todo. Ou inserimos nas duas tabelas ou em nenhuma, mantendo assim a integridade dos dados no banco. A mesma coisa deve ser feita para as alterações e exclusões.

Para executarmos as operações de inserção, alteração e exclusão nas duas tabelas podemos utilizar dois componentes *UpdateSQL*. Assim, não será mais possível utilizar a propriedade **UpdateObject** do componente *TQuery* para definir qual dos dois componentes será utilizado, já que queremos utilizar os dois.

Devemos então utilizar, ao invés dessa propriedade, um evento do componente *TQuery* chamado **OnUpdateRecord**. Através desse evento, podemos chamar cada um dos comandos fornecidos pelos componentes *UpdateSQL* e fazer todas as atualizações necessárias nas tabelas.



Fig 8.12: Data Module.

```
procedure TDMSistVendas.QClienteUpdateRecord(DataSet: TDataSet;  
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);  
begin  
  If Not Assigned(USPessoa.DataSet) then begin  
    USPessoa.DataSet:=TBDEDataSet(DataSet);  
    USCliente.DataSet:=TBDEDataSet(DataSet);  
  end;  
  if UpdateKind = ukDelete then begin  
    USCliente.Apply(UpdateKind);  
    USPessoa.Apply(UpdateKind);  
  end else begin  
    USPessoa.Apply(UpdateKind);  
    USCliente.Apply(UpdateKind);  
  end;  
  UpdateAction:=uaApplied;  
end;
```

Na implementação, devemos primeiramente inicializar a propriedade **DataSet** dos componentes *UpdateSQL* caso seja a primeira vez que o código esteja sendo executado.

A seguir é utilizado o método **Apply** dos componentes *UpdateSQL* para executar o comando SQL de acordo com a informação passada pelo BDE de qual operação está sendo realizada. Se a operação for de exclusão, iremos excluir primeiro na tabela de CLIENTE e depois na tabela de PESSOA para manter a integridade. E o contrário é feito no caso de inserções e alterações de registros.

Se a operação for realizada como sucesso, iremos definir o parâmetro **UpdateAction** como sendo “uaApplied” que permite que mais tarde o registro seja retirado do “cached update” através do comando **CommitUpdates**. Se algum erro ocorrer define o parâmetro como “usFail” e continua a propagação do erro para que o controle de transação feito na tela de cliente possa fazer o “rollback” no banco de dados.

Devemos agora fazer algumas modificações na tela de manutenção de clientes para implementar o controle de transação. Primeiro vamos criar uma procedure nova para cuidar da habilitação dos botões. Retira-se o código que estava no evento **OnStateChange** do *DataSource* e o coloca nessa nova procedure com algumas modificações.

```
procedure TfrmCliente.EnableBtnControls;  
begin  
  Case DSMain.State of  
    dsInactive:  
      begin  
        btnAppend.Enabled:=TRUE;  
        btnDelete.Enabled:=FALSE;  
        btnSave.Enabled:=FALSE;  
        btnDiscard.Enabled:=FALSE;  
        btnSearch.Enabled:=TRUE;  
      end;  
    dsInsert:  
      begin  
        btnAppend.Enabled:=FALSE;  
        btnDelete.Enabled:=FALSE;  
        btnSave.Enabled:=TRUE;  
        btnDiscard.Enabled:=TRUE;  
        btnSearch.Enabled:=FALSE;  
      end;  
    dsEdit:  
      begin  
        btnAppend.Enabled:=FALSE;
```

```
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=TRUE;
        btnDiscard.Enabled:=TRUE;
        btnSearch.Enabled:=FALSE;
    end;
dsBrowse:
    If TQuery(DSMain.DataSet).UpdatesPending then begin
        btnAppend.Enabled:=FALSE;
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=TRUE;
        btnDiscard.Enabled:=TRUE;
        btnSearch.Enabled:=FALSE;
    end else begin
        btnAppend.Enabled:=TRUE;
        btnDelete.Enabled:=TRUE;
        btnSave.Enabled:=FALSE;
        btnDiscard.Enabled:=FALSE;
        btnSearch.Enabled:=TRUE;
    end;
end;
end;
```

Agora podemos modificar os eventos de gravação e exclusão:

```
procedure TfrmCliente.btnDeleteClick(Sender: TObject);
begin
    DMSistVendas.Databasel.StartTransaction;
    try
        DSMain.DataSet.Delete;
        TBDEDataSet(DSMain.DataSet).ApplyUpdates;
        DMSistVendas.Databasel.Commit;
        TBDEDataSet(DSMain.DataSet).CommitUpdates;
        DSMain.DataSet.close;
    except
        DMSistVendas.Databasel.Rollback;
        TBDEDataSet(DSMain.DataSet).CancelUpdates;
    end;
end;

procedure TfrmCliente.btnSaveClick(Sender: TObject);
begin
    DMSistVendas.Databasel.StartTransaction;
    try
        TBDEDataSet(DSMain.DataSet).ApplyUpdates;
        DMSistVendas.Databasel.Commit;
        TBDEDataSet(DSMain.DataSet).CommitUpdates;
        EnableBtnControls;
    except
        DMSistVendas.Databasel.Rollback;
    end;
end;
```

Mantendo o Result Set da Consulta

Toda vez que a tela de consulta de clientes é aberta o *DataSet* está fechado para permitir uma nova pesquisa do usuário. No entanto, pode-se optar em manter o último “result set” feito quando se abrir a tela novamente, supondo que o usuário tenha uma grande chance de encontrar o próximo registro para manutenção no próprio conjunto de linhas retornado pela pesquisa anterior.

Para fazer isso devemos alterar o código da tela de consulta para que ela não feche o *DataSet* quando o form for destruído. Além disso algumas outras alterações devem ser feitas.

Devemos alterar os seguintes eventos do form:

```
procedure TfrmConsCliente.FormCreate(Sender: TObject);
begin
  // CmdSelect:=TQuery(DSMain.DataSet).SQL.Text;
  CmdSelect:=DMSistVendas.CmdSelectQCliente;
  If Not DSUF.DataSet.Active then DSUF.DataSet.Open;
end;

procedure TfrmConsCliente.FormDestroy(Sender: TObject);
begin
  //DSCidade.DataSet.close;
  //TQuery(DSMain.DataSet).SQL.Text:=CmdSelect;
end;
```

Devemos também declarar e inicializar a variável **CmdSelectQCliente** no *DataModule* para armazenar o comando original da “query”.

```
private
  { Private declarations }
public
  { Public declarations }
  CmdSelectQCliente: String;
end;

procedure TDMSistVendas.DMSistVendasCreate(Sender: TObject);
begin
  CmdSelectQCliente:=QConsCliente.SQL.Text;
end;
```

Fazendo essas alterações, quando se abre novamente a tela a última consulta feita pelo usuário permanece ativa. Entretanto, deve-se ter cuidado com essa abordagem. Se o “result set” permanecer aberto e esse possuir muitas linhas que ainda não foram trazidas para aplicação, essas podem ser trazidas quando o usuário realizar uma manutenção no registro e salvar as alterações. Como foi visto, para alguns bancos de dados o Delphi realiza a busca prematura dos registro de um “result set” pendente antes de realizar o **commit**.

Portanto, deve-se optar pelo melhor desempenho e funcionalidade para o usuário de acordo com o banco de dados. As opções para os bancos que não suportam manter um cursor aberto após o fechamento da transação seriam:

- Tentar minimizar o número de registros que serão trazidos na consulta forçando o usuário a atribuir algum tipo de filtro.
- Utilizar uma outra conexão, através de um outro componente *Database* para se fazer a consulta, desta forma a finalização da transação de uma conexão não interfere na outra. Entretanto, essa opção costuma consumir um número maior de licenças de usuários do banco de dados. Normalmente, cada conexão corresponde ao uso de uma licença.

- Tentar utilizar na tela de consulta um componente *TTable* ao invés do componente *TQuery*. Como foi visto o componente *TTable* possui um comportamento mais inteligente no caso do fechamento da transação, já que permite que o cursor seja fechado e quando houver a necessidade de caminhar por linhas ainda não trazidas dispara um novo comando **select**. Entretanto, esse componente costuma abrir uma transação e pode estar mantendo algum registro travado, dependendo do banco de dados utilizado.

Controlando Transações Master/Detail

Esse capítulo mostra como implementar telas que exigem um controle de transação do tipo Master/Detail.

Através do recurso de controle de transações disponíveis nos servidores de banco de dados, é possível construir aplicações que mantenham a unicidade lógica de um conjunto de operações. Muitas vezes é necessário que as atualizações em várias tabelas sejam feitas em conjunto para que a integridade dos dados seja preservada. Se qualquer problema for encontrado durante a transação, os dados devem retornar aos seus valores originais antes do início da transação.

Em algumas aplicações, para manter a integridade dos dados através do controle de transação é necessário inserir um registro de uma tabela principal juntamente com vários outros registros relacionados de uma segunda tabela. É o caso por exemplo de uma tela de entrada de pedidos. Em um cadastro de pedido é inserido dados do cabeçalho (número, cliente, vendedor, data) e dados de cada item que está sendo vendido. É comum que as regras de negócio que a aplicação deva seguir, imponha que o pedido deva ser inserido como um todo (cabeçalho e seus itens), ou seja, em um única transação.

A esse tipo de tela é dado o nome de Master/Detail, ou seja, a tabela mestre e seus detalhes. No exemplo de pedido, a entidade master é representada pela tabela de PEDIDO e a entidade detalhe é representada pela tabela de ITEM.

Tela de Consulta

Podemos começar definindo nossa tela de consulta, que o usuário irá utilizar para pesquisar um pedido, assim como foi feito para a tela de clientes.

Utilizaremos mais três componentes *TQuery* que serão colocados no *Data Module*.

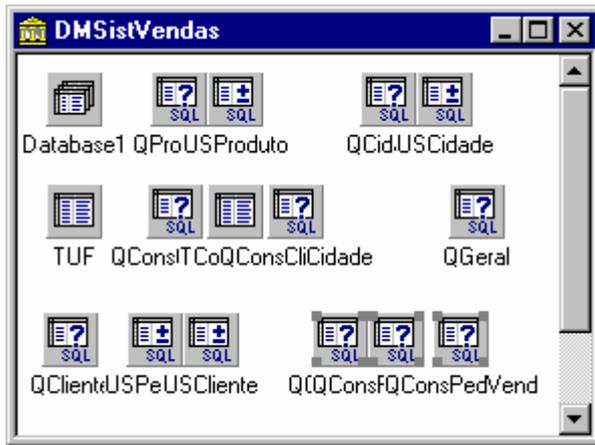


Fig: 9.1: Data Module.

Componente	Propriedade	Valor
Query1	Name	QConsPed
	DatabaseName	DBVENDAS
	SQL	SELECT PEDIDO.PED_CD , PEDIDO.PED_DT , PEDIDO.PED_VALOR , PEDIDO.PED_TIPO FROM PEDIDO PEDIDO ORDER BY PEDIDO.PED_CD
Query2	Name	QConsPedCliente
	DatabaseName	DBVENDAS
	SQL	SELECT PESSOA.PES_CD , PESSOA.PES_NM FROM CLIENTE CLIENTE , PESSOA PESSOA WHERE (CLIENTE.PES_CD = PESSOA.PES_CD) ORDER BY PESSOA.PES_NM *Obs: Deixe a quarta linha em branco
Query3	Name	QConsPedVend
	DatabaseName	DBVENDAS
	SQL	SELECT PESSOA.PES_CD , PESSOA.PES_NM FROM VENDEDOR VENDEDOR , PESSOA PESSOA WHERE (VENDEDOR.PES_CD = PESSOA.PES_CD) ORDER BY PESSOA.PES_NM

Tabela de Propriedades

Devemos, agora, montar a tela como a figura a seguir:

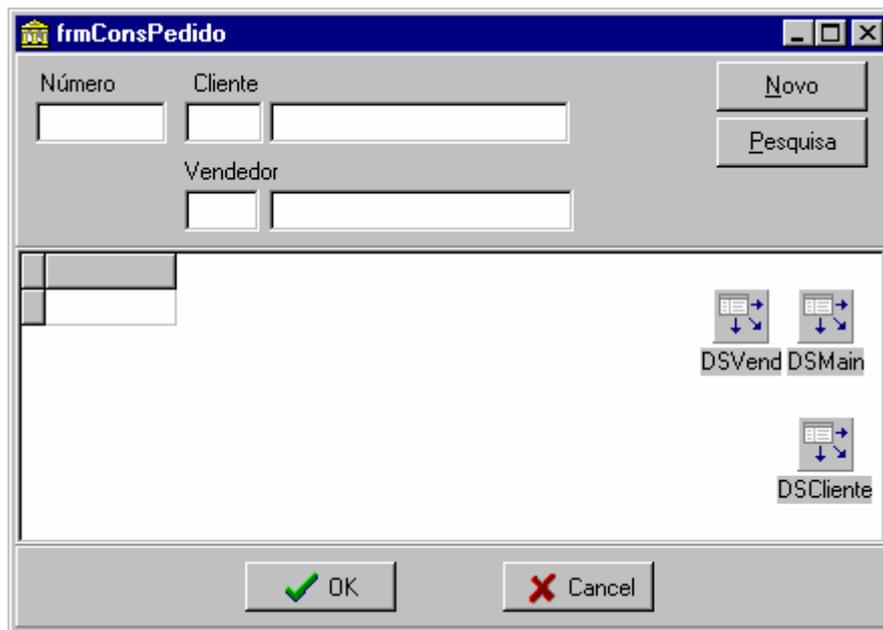


Fig 9.2: Tela de Consulta de Pedidos.

Componente	Propriedade	Valor
form1	Name	frmConsPedido
	Caption	Pedidos
DataSource1	Name	DSMain
	DataSet	DMSistVendas.QConsPed
DataSource2	Name	DSCliente
	DataSet	DMSistVendas.QConsPedCliente
DataSource3	Name	DSVend
	DataSet	DMSistVendas.QConsPedVend
Panel1	Align	alTop
Edit...	Name	edtNumero, edtCodCliente, edtNomeCliente, edtCodVend, edtNomeVend
button1	Name	btnPesquisa
	Caption	Pesquisa
button2	Name	btnNovo
	Caption	Novo
Panel2	Align	alClient
DBGrid1	Align	alClient
	DataSource	DSMain
Panel3	Align	alBottom
bitbtn1	Kind	bkOk
bitbtn2	Kind	bkCancel

Tabela de Propriedades

Iremos propor aqui uma consulta um pouco diferente da apresentada na tela de consulta de clientes e vendedores. Para escolher um cliente, o usuário poderá digitar as primeiras letras do nome, antes de abrir o Grid para consulta, tornando a pesquisa mais seletiva. Desta forma evitaremos trazer todos os registros da tabela de cliente a estação de trabalho, além de permitir uma localização mais rápida e precisa do registro pelo usuário. Com isso o usuário utiliza melhor e se beneficia dos recursos fornecidos pelos servidores de banco de dados.

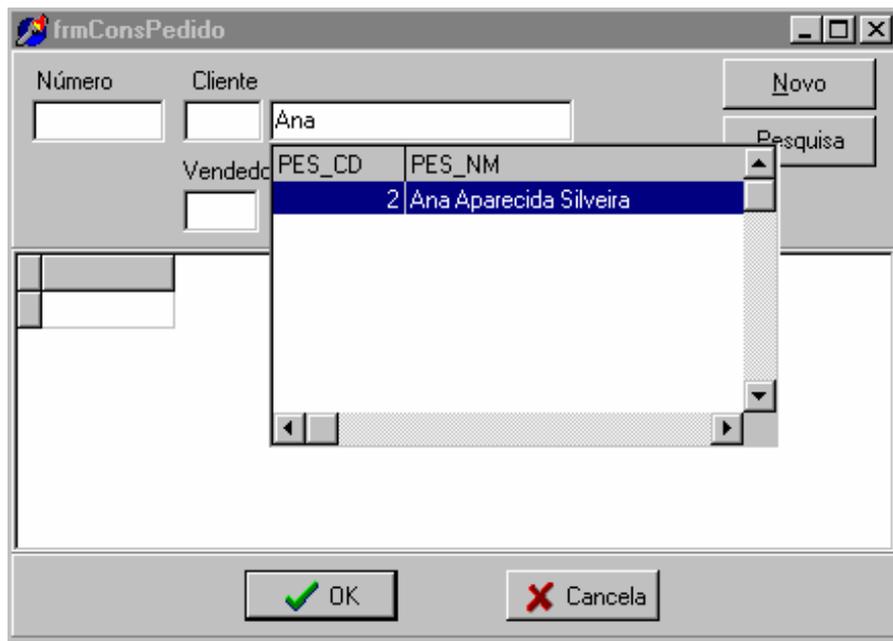


Fig 9.3: Pesquisa de cliente filtrada pelas letras digitadas pelo usuário.

A seguir apresentamos o código da “unit” da tela de consulta de pedidos:

```

unit conspedido;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Grids, DBGrids, ExtCtrls, Db, Buttons;

type
  TfrmConsPedido = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    edtNumero: TEdit;
    Label1: TLabel;
    Label4: TLabel;
    edtCodCliente: TEdit;
    edtNomeCliente: TEdit;
    Label5: TLabel;
    edtNomeVend: TEdit;
    DBGrid1: TDBGrid;
    btnPesquisa: TButton;
    btnNovo: TButton;
    edtCodVend: TEdit;
    Panel3: TPanel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    DSMain: TDataSource;
    DSCliente: TDataSource;
    DSVend: TDataSource;
    procedure btnNovoClick(Sender: TObject);
    procedure btnPesquisaClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure edtNomeClienteKeyUp(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure edtNomeVendKeyUp(Sender: TObject; var Key: Word;
      Shift: TShiftState);
  end;

```

```

private
  { Private declarations }
public
  { Public declarations }
  CmdSelect:String;
end;

var
  frmConsPedido: TfrmConsPedido;

implementation

uses Dm01,dbtables, ConsGrid;

{$R *.DFM}

procedure TfrmConsPedido.btnNovoClick(Sender: TObject);
var i:integer;
begin
  For i:=0 to Panell.ControlCount - 1 do
    If Panell.Controls[i] is TEdit then
      (Panell.Controls[i] as TEdit).Clear;
  end;

procedure TfrmConsPedido.btnPesquisaClick(Sender: TObject);
var sWhere,sSelect,sep:string;
    nPosOrderBy:integer;
begin
  TQuery(DSMain.DataSet).Close;
  Sep:='';
  sWhere:=' where ';
  sSelect:=CmdSelect;
  If (edtNumero.Text <> '') then begin
    sWhere:=Format('%s %s (%s = %s) ',[sWhere,Sep,'PED_CD',edtNumero.Text]);
    Sep:='And';
  end;
  If (edtCodCliente.Text <> '') then begin
    sWhere:=Format('%s %s (%s = %s)',
      [sWhere,Sep,'PES_CD_CLI',edtCodCliente.Text]);
    Sep:='And';
  end;
  If (edtCodVend.Text <> '') then begin
    sWhere:=Format('%s %s (%s = %s)',
      [sWhere,Sep,'PES_CD_VEN',edtCodVend.Text]);
    Sep:='And';
  end;
  If Sep <> '' then begin
    nPosOrderBy:=Pos('ORDER BY', UpperCase(sSelect));
    if nPosOrderBy = 0 then
      sSelect:=sSelect + sWhere
    else
      Insert(sWhere,sSelect,nPosOrderBy);
  end;
  TQuery(DSMain.DataSet).SQL.Text:=sSelect;
  TQuery(DSMain.DataSet).Open;
end;

procedure TfrmConsPedido.FormCreate(Sender: TObject);
begin
  CmdSelect:=TQuery(DSMain.DataSet).SQL.TExt;
end;

procedure TfrmConsPedido.FormDestroy(Sender: TObject);
begin
  DSMain.DataSet.Close;
  TQuery(DSMain.DataSet).SQL.TExt:=CmdSelect;
end;

```

```

procedure TfrmConsPedido.edtNomeClienteKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  If Key = 113 then begin
    DSCliente.DataSet.Close;
    If edtNomeCliente.Text <> '' then
      TQuery(DSCliente.DataSet).SQL[3]:= ' AND PESSOA.PES_NM LIKE ''
        + edtNomeCliente.Text + '%''
    Else
      TQuery(DSCliente.DataSet).SQL[3]:= '';
    DSCliente.DataSet.Open;
    frmGrid:=TFrmGrid.Create(Self);
    frmGrid.DBGrid1.DataSource:=DSCliente;
    If (frmGrid.ShowModal=mrOk) and (Not DSCliente.DataSet.IsEmpty)
    then begin
      TEdit(Sender).Text:=DSCliente.DataSet['PES_NM'];
      edtCodCliente.Text:=DSCliente.DataSet.FieldByName('PES_CD').AsString;
    end;
  end;
end;

procedure TfrmConsPedido.edtNomeVendKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  If Key = 113 then begin
    If Not DSVend.DataSet.Active then
      DSVend.DataSet.Open;
    frmGrid:=TFrmGrid.Create(Self);
    frmGrid.DBGrid1.DataSource:=DSVend;
    If (frmGrid.ShowModal=mrOk) and (Not DSVend.DataSet.IsEmpty)
    then begin
      TEdit(Sender).Text:=DSVend.DataSet['PES_NM'];
      edtCodVend.Text:=DSVend.DataSet.FieldByName('PES_CD').AsString;
    end;
  end;
end;

end.

```

Tela de Manutenção

Para fazer a manutenção do pedido, vamos acrescentar no *Data Module* os seguintes componentes:

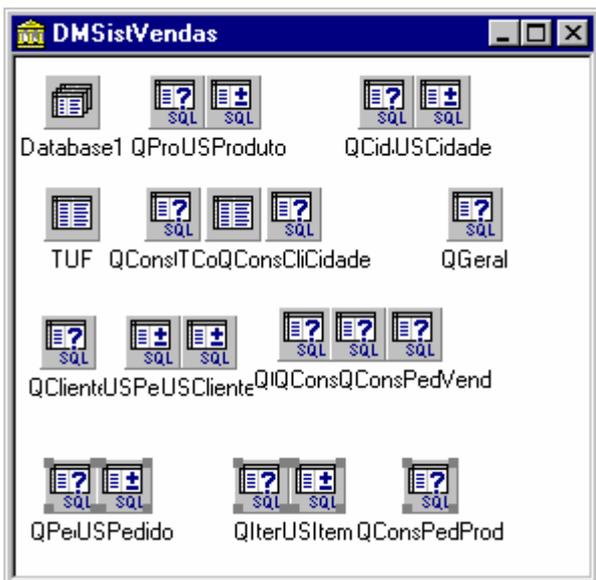


Fig. 9.4: Data Module

Componente	Propriedade	Valor
Query1	Name	Qpedido
	DatabaseName	DBVENDAS
	SQL	SELECT PEDIDO.PED_CD , PEDIDO.PED_DT , PEDIDO.PED_VALOR , PEDIDO.PED_TIPO , PEDIDO.PES_CD_CLI , PEDIDO.PES_CD_VEN , PESSOA.PES_NM FROM PEDIDO PEDIDO , CLIENTE CLIENTE , PESSOA PESSOA WHERE (CLIENTE.PES_CD = PESSOA.PES_CD) AND (PEDIDO.PES_CD_CLI = CLIENTE.PES_CD) AND (PEDIDO.PED_CD = :PED_CD) ORDER BY PEDIDO.PED_CD
	UpdateObject	USPedido
Query2	UpdateObject	USPedido
	CachedUpdate	TRUE
	Name	Qitem
	DatabaseName	DBVENDAS
UpdateSQL1	SQL	SELECT ITEM.PED_CD , ITEM.PRO_CD , ITEM.ITE_VALOR , ITEM.ITE_QUANT , PRODUTO.PRO_NM FROM ITEM ITEM , PRODUTO PRODUTO WHERE (ITEM.PRO_CD = PRODUTO.PRO_CD) and (ITEM.PED_CD = :PED_CD) ORDER BY ITEM.PRO_CD
	UpdateObject	USItem
	CachedUpdate	TRUE
	Name	USPedido
UpdateSQL2	Name	USItem
	Name	QConsPedProd
Query3	DatabaseName	DBVENDAS
	SQL	SELECT * FROM PRODUTO ORDER BY PRO_NM
		Obs: Deixar a segunda linha em branco.

Tabela de Propriedades

Deve-se criar também, um novo campo do tipo “lookup” em *QPedido* para permitir a escolha do vendedor através de seu nome. Iremos nessa tela, utilizar o próprio recurso de “lookup” do Delphi para a seleção do vendedor.

A tela de manutenção de pedidos deverá ser construída como a figura abaixo. Pode-se notar que serão apresentadas na mesma tela os dados das tabelas PEDIDO e ITEM para permitir a manutenção integral dos dados do pedido.

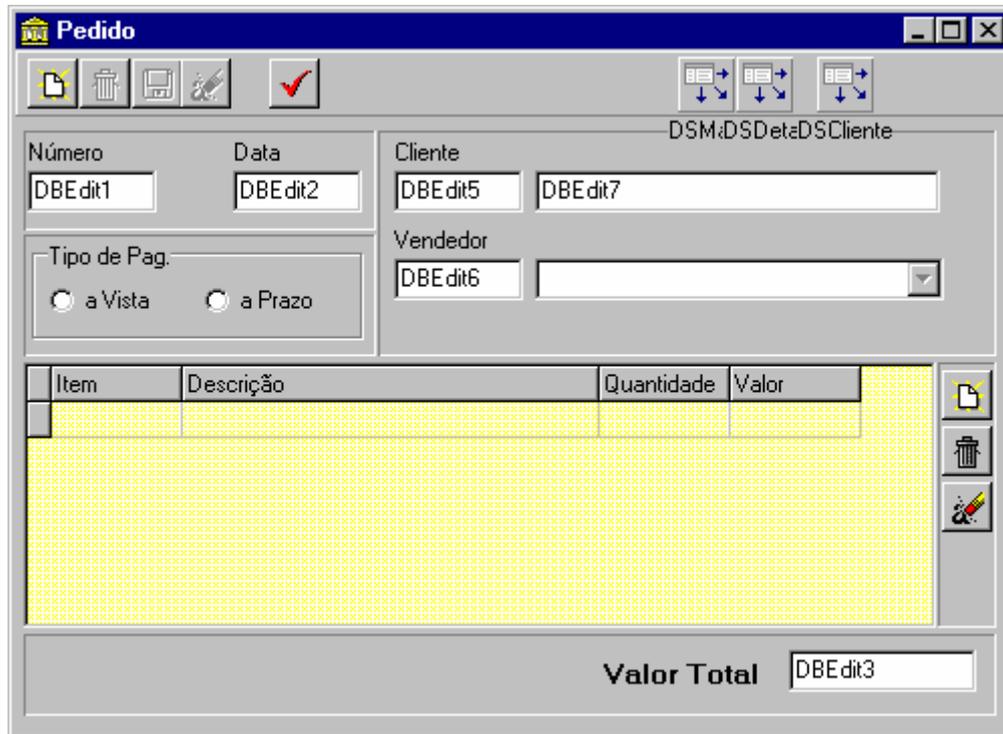


Fig 9.5: Tela de Manutenção de Pedido.

Componente	Propriedade	Valor
form1	Name	frmPedido
	Caption	Pedidos
DataSource1	Name	DSMain
	DataSet	DMSistVendas.Qpedido
DataSource2	Name	DSDetail
	DataSet	DMSistVendas.QItem
DataSource3	Name	DSCliente
	DataSet	DMSistVendas.QConsPedCliente
Panel1	Align	alTop
SpeedButton1	Name	btnAppend
SpeedButton2	Name	btnDelete
SpeedButton3	Name	btnSave
SpeedButton4	Name	btnDiscard
SpeedButton5	Name	btnSearch
DBEdit..., DBRadioGroup	DataSource	DSMain
DBGrid1	DataSource	DSDetail
SpeedButton6	Name	btnDetailAppend
SpeedButton7	Name	btnDetailDelete
SpeedButton8	Name	btnDetailDiscard

Tabela de Propriedades

Lógica visual

Antes de começar a definir os eventos dos botões, iremos criar um procedimento para controlar a habilitação desses botões.

```

procedure TfrmPedido.EnableBtnControls;
begin
  Case DSMain.State of
    dsInactive:
      begin
        btnAppend.Enabled:=TRUE;
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=FALSE;
        btnDiscard.Enabled:=FALSE;
        btnSearch.Enabled:=TRUE;
      end;
    dsInsert:
      begin
        btnAppend.Enabled:=FALSE;
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=TRUE;
        btnDiscard.Enabled:=TRUE;
        btnSearch.Enabled:=FALSE;
      end;
    dsEdit:
      begin
        btnAppend.Enabled:=FALSE;
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=TRUE;
        btnDiscard.Enabled:=TRUE;
        btnSearch.Enabled:=FALSE;
      end;
    dsBrowse:
      If TQuery(DSMain.DataSet).UpdatesPending then begin
        btnAppend.Enabled:=FALSE;
        btnDelete.Enabled:=FALSE;
        btnSave.Enabled:=TRUE;
        btnDiscard.Enabled:=TRUE;
        btnSearch.Enabled:=FALSE;
      end else begin
        btnAppend.Enabled:=TRUE;
        btnDelete.Enabled:=TRUE;
        btnSave.Enabled:=FALSE;
        btnDiscard.Enabled:=FALSE;
        btnSearch.Enabled:=TRUE;
      end;
  end;
end;

```

Podemos também definir alguns eventos de comportamento geral:

```

procedure TfrmPedido.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  DSDetail.DataSet.Close;
  DSMain.DataSet.Close;
  DSCliente.DataSet.close;
end;

```

O evento **OnClose** do *Form* pode ser utilizado para fechar os *DataSets* que serão abertos durante o uso da tela.

Através do evento **OnStateChange**, podemos chamar a *procedure* **EnableBtnControls** para controlar a habilitação dos botões.

```
procedure TfrmPedido.DSMainStateChange(Sender: TObject);
begin
    EnableBtnControls;
end;
```

Agora vamos implementar os eventos de cada botão da tela, a começar pelos da Detail (tabela de itens).

No evento **OnClick** do botão *btnDetailAppend* da Detail, podemos implementar a abertura de uma nova linha no *Grid* através do comando **Append**.

```
procedure TfrmPedido.btnDetailAppendClick(Sender: TObject);
begin
    DSDetail.DataSet.Append;
end;
```

Além disso, devemos implementar o evento **OnStateChange** do *DataSource* Detail para podermos notificar a Master de qualquer alteração feita na Detail:

```
procedure TfrmPedido.DSDetailStateChange(Sender: TObject);
begin
    If DSDetail.State in [dsInsert,dsEdit] then
        DSMain.DataSet.Edit;
end;
```

O botão de “Delete” pode ser implementado da seguinte forma:

```
procedure TfrmPedido.btnDetailDeleteClick(Sender: TObject);
begin
    If TBDEDataSet(DSDetail.DataSet).UpdateStatus <> usDeleted then
        begin
            DSMain.DataSet.Edit;
            DSDetail.DataSet.Delete;
        end;
end;
```

A intenção do botão *btnDetailDelete* é simplesmente marcar a linha para exclusão. A exclusão no banco de dados só será feita quando o usuário mandar salvar todas as alterações.

A propriedade **UpdateStatus** indica o estado da linha no “cached updates”. Os estados podem ser: não modificada, modificada, inserida ou alterada. No evento acima estamos verificando se a linha já não está marcada para exclusão, para evitarmos de tentar excluir a linha novamente.

Para sensibilizar a Master da alteração feita na Detail, podemos chamar o método **Edit** do *DataSet*. Desta forma os botões de salvamento e cancelamento associados a Master serão habilitados.

E por último devemos chamar o método **Delete** do *DataSet* para marcar a linha para exclusão mandando-a para o “cached updates”.

O botão *btnDetailDiscard* serve para descartar as alterações feitas em uma determinada linha. Se as alterações ainda não foram enviadas para o “cached updates”, a operação de Discard simplesmente cancela as alterações através do método **Cancel**. Mas, se a linha já foi enviada, esta deverá ser revertida a seu estado original através do método **RevertRecord**, sendo portanto retirada do “cached updates”.

```

procedure TfrmPedido.btnDetailDiscardClick(Sender: TObject);
begin
  If DSDetail.State in [dsEdit,dsInsert] then begin
    DSDetail.DataSet.Cancel
  end else begin
    TBDEDataSet(DSDetail.DataSet).RevertRecord;
  end;
end;

```

Podemos fornecer um efeito visual para possibilitar o usuário identificar as linhas alteradas, excluídas e inseridas. Para fazer isso, podemos implementar o evento **OnDrawColumnCell** do *Grid* e alterar as cores do fonte de cada uma dessas situações. Para isso podemos utilizar a propriedade **UpdateStatus** do *DataSet*.

```

procedure TfrmPedido.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
var OldColor:Tcolor;
begin
  with TDBGrid(Sender) do
    //Atribui cores diferentes para cada linha da tabela de acordo com seu
    status
    case TQuery(DSDetail.DataSet).UpdateStatus of
      usDeleted:
        begin
          OldColor:=Canvas.Font.Color;
          Canvas.Font.Color:=clRed;
          Canvas.Font.Style:=Canvas.Font.Style + [fsStrikeOut];
          DefaultDrawColumnCell(Rect,DataCol,Column,State);
          Canvas.Font.Color:=OldColor;
          Canvas.Font.Style:=Canvas.Font.Style - [fsStrikeOut];
        end;
      usModified:
        begin
          OldColor:=Canvas.Font.Color;
          Canvas.Font.Color:=clBlue;
          DefaultDrawColumnCell(Rect,DataCol,Column,State);
          Canvas.Font.Color:=OldColor;
        end;
      usInserted:
        begin
          OldColor:=Canvas.Font.Color;
          Canvas.Font.Color:=clGreen;
          DefaultDrawColumnCell(Rect,DataCol,Column,State);
          Canvas.Font.Color:=OldColor;
        end;
    end;
  end;
end;

```

Antes de implementarmos os eventos da Master, vamos criar um procedimento que nos permita pesquisar os registro da Detail de acordo com o registro selecionado na Master.

```

procedure TfrmPedido.DetailSearch;
begin
  DSDetail.DataSet.close;
  If DSMain.DataSet.FieldByName('PED_CD').IsNull then
    TQuery(DSDetail.DataSet).Params[0].clear
  else
    TQuery(DSDetail.DataSet).Params[0].value:=
      DSMain.DataSet['PED_CD'];
  DSDetail.DataSet.Open;
  TBDEDataSet(DSDetail.DataSet).UpdateRecordTypes:=
    TBDEDataSet(DSDetail.DataSet).UpdateRecordTypes + [rtDeleted];
end;

```

A última linha do código permite que os registros excluídos continuem visíveis no *DataSet* para que possam ser revertidos conforme a necessidade do usuário.

Podemos começar a implementação dos eventos da Master através do botão *btnAppend*.

```

procedure TfrmPedido.btnAppendClick(Sender: TObject);
begin
  If Not DSMain.DataSet.Active then begin
    DSMain.DataSet.Open;
  end;
  DSMain.DataSet.Append;
  DetailSearch;
end;

```

Esse evento verifica se o *DataSet* Master está aberto e a seguir insere uma nova linha através do comando **Append**. Depois é feita uma pesquisa na tabela Detail com o parâmetro nulo para que nenhum registro seja encontrado e permita assim a inserção de novos registros no Grid de Itens.

A pesquisa da Master realizada pelo botão *btnSearch* pode ser implementada da seguinte forma

```

procedure TfrmPedido.btnSearchClick(Sender: TObject);
begin
  DSMain.DataSet.Close;
  frmConsPedido:=TfrmConsPedido.Create(Self);
  If (frmConsPedido.ShowModal=mrOk)
    and (Not (frmConsPedido.DSMain.DataSet['PED_CD'] = Null)) then
  begin
    TQuery(DSMain.DataSet).Params[0].value:=
      frmConsPedido.DSMain.DataSet['PED_CD'];
    DSMain.DataSet.Open;
    DetailSearch;
  end;
  frmConsPedido.free;
end;

```

O botão *btnDiscard* é responsável pelo cancelamento das alterações da Master. Apesar da Master estar tratando uma única linha de cada vez, não é suficiente verificarmos o estado do *DataSource* para sabermos se a linha possui alterações. Quando uma tentativa não bem sucedida de salvar o registro acontece, o registro foi colocado no “cached updates”, portanto seu estado será “dsBrowse”. Por isso é necessário verificarmos a propriedade **UpdateStatus** para sabermos como descartar as alterações.

```

procedure TfrmPedido.btnDiscardClick(Sender: TObject);
begin
  If TBDEDataSet(DSMain.DataSet).UpdateStatus in [usModified,
    usInserted, usDeleted] then
    TBDEDataSet(DSMain.DataSet).RevertRecord
  else
    TBDEDataSet(DSMain.DataSet).Cancel;
  If TBDEDataSet(DSMain.DataSet).IsEmpty then begin
    DSMain.DataSet.close;
    DSDetail.DataSet.Close;
  end else begin
    DetailSearch;
  end;
  EnableBtnControls;
end;

```

Controle visual da Transação

O evento de salvamento feito pelo botão *btnSave* pode ser implementado da seguinte forma para manter em uma única transação a inserção do pedido e de seus itens.

```

procedure TfrmPedido.btnSaveClick(Sender: TObject);
begin
  If DSDetail.DataSet.State in [dsInsert,dsEdit] then
    DSDetail.DataSet.Post;
  If DSMain.DataSet.State in [dsInsert,dsEdit] then
    DSMain.DataSet.Post;
  DMSistVendas.Databasel.StartTransaction;
  try
    TBDEDataSet(DSMain.DataSet).ApplyUpdates;
    TBDEDataSet(DSMain.DataSet).CommitUpdates;
    DMSistVendas.Databasel.Commit;
    EnableBtnControls;
    DetailSearch;
  except
    On E:Exception do begin
      TBDEDataSet(DSMain.DataSet).CommitUpdates;
      DMSistVendas.Databasel.Rollback;
      DSMain.DataSet.Edit;
      if Not ((E is EDBEngineError) and
        (EDBEngineError(E).Errors[0].NativeError = 0)) then
        raise;
      end;
    end;
  end;
end;

```

Desta forma se algum problema ocorrer na inserção do pedido ou de seus itens é executado um **rollback** no banco de dados para desfazer toda a transação e a tela fica intacta para que o usuário possa corrigir o problema e refazer a gravação.

A primeira parte do código cuida de enviar para o “cached update” qualquer linha que esteja pendente na Detail ou na Master. A linha da Detail é enviada primeiro, porque ela pode causar modificações no registro corrente da Master.

```

  If DSDetail.DataSet.State in [dsInsert,dsEdit] then
    DSDetail.DataSet.Post;
  If DSMain.DataSet.State in [dsInsert,dsEdit] then
    DSMain.DataSet.Post;

```

A seguir começa a transação e toda a operação de salvamento é feita dentro do comando **try..except** para que a finalização da transação possa ser controlada, tanto no caso de sucesso como no caso de ocorrer alguma falha. O salvamento é feito através do comando **ApplyUpdates**, responsável por enviar as pendências do “cached update” para o banco. Após o envio com sucesso, é necessário executar o comando **CommitUpdates**, responsável em retirar do “cached update” as linhas salvas com sucesso. Entretanto, para que o Delphi atualize alguns flags, esse comando é necessário mesmo se a atualização falhar.

Se essas operações ocorrerem com sucesso, é executado o comando **commit** no banco de dados e todo o conjunto de atualizações é finalmente aplicado no banco, caso contrário toda a transação é desfeita através do comando **rollback**.

O código ainda apresenta alguns detalhes como: chamar o controle de habilitação dos botões e fazer a busca novamente dos dados da Detail para que os registros possam ser atualizados na tela.

Transação - Lógica de negócio

Alguns eventos devem ser implementados no *DataModule* para que o controle da transação Master/Detail seja feito por completo.

Vamos começar pelo evento que deve ser utilizado para fornecer valores “default” para os campos, quando o usuário criar um novo registro:

```
procedure TDMSistVendas.QPedidoNewRecord(DataSet: TDataSet);
begin
  DataSet['PED_VALOR']:=0;
  DataSet['PED_TIPO']:= 'V';
end;
```

Agora vamos implementar o evento **UpdateRecord** do *DataSet QPedido*.

```
procedure TDMSistVendas.QPedidoUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  USPedido.Apply(UpdateKind);
  try
    QItem.ApplyUpdates;
  finally
    QItem.CommitUpdates;
  end;
  UpdateAction:=uaApplied;
end;
```

Esse evento simplesmente cuida de executar o comando de atualização através do componente *UpdateSQL* e fazer a chamada para a efetivação do “cached update” da Detail. Desta forma se algum erro acontecer na Master ou na Detail, o valor do flag retornado através do parâmetro **UpdateAction** é “uaAbort”, o que faz a linha da Master continuar no cache para novas tentativas de atualização pelo usuário. Quando se retorna o valor “uaApplied” no parâmetro **UpdateAction**, o Delphi reverte o status da linha para não modificada como se ela tivesse sido aplicada no banco. Isso só pode ser feito se toda a transação foi realizada com sucesso e tivermos certeza que o comando **commit** será executado.

Devemos também implementar o evento da Detail.

```
procedure TDMSistVendas.QItemUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  USItem.Apply(UpdateKind);
  UpdateAction:=uaSkip;
end;
```

Para evitarmos que os flags dos registros sejam revertidos para não modificado, foi retornado através do parâmetro **UpdateAction** o valor “uaSkip” que não produz nenhuma exceção e não modifica o flag dos registros. Quando a transação é finalizada com sucesso, é feita uma nova pesquisa no próprio evento do botão *btnSave* para que todos os flags sejam atualizados.

Desta forma conseguimos manter a integridade da transação e permitir a correção dos registros pelo usuário para que possam ser reenviados para o banco de dados.

Calculando o Valor Total do Pedido

Voltando a tela de pedido, podemos calcular o valor do pedido sempre que houver uma atualização no valor do item. O valor do item é também função de dois outros valores: o preço do produto e a quantidade do item.

Vamos utilizar o evento **OnDataChange** do *DataSource* Detail para fazermos os cálculos.

```
procedure TfrmPedido.DSDetailDataChange(Sender: TObject; Field: TField);
begin
  If (Field = DSDetail.DataSet.FieldByName('PRO_CD')) or
    (Field = DSDetail.DataSet.FieldByName('ITE_QUANT')) then begin
    If DSMain.State = dsBrowse then DSMain.DataSet.Edit;
    DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
      DSMain.DataSet.FieldByName('PED_VALOR').asfloat-
      DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
    DSDetail.DataSet['ITE_VALOR']:=
      DSDetail.DataSet['PRO_PRECO'] * DSDetail.DataSet['ITE_QUANT'];
    DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
      DSMain.DataSet.FieldByName('PED_VALOR').asfloat +
      DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
  end;
end;
```

Além disso devemos recalcular o valor total em várias situações de alteração que podem acontecer na Detail. Por isso, iremos adicionar algumas linhas de código em alguns dos eventos já implementados anteriormente.

O primeiro é o evento do botão *btnDetailDelete* que deve subtrair o valor do item do total do pedido.

```
procedure TfrmPedido.btnDetailDeleteClick(Sender: TObject);
begin
  If TBDEDataSet(DSDetail.DataSet).UpdateStatus <> usDeleted then begin
    DSMain.DataSet.Edit;
    DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
      DSMain.DataSet.FieldByName('PED_VALOR').asfloat -
      DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
    DSDetail.DataSet.Delete;
  end;
```

end;

Além disso, devemos também recalculer o valor do pedido no evento do botão *btnDetailDiscard*.

```

procedure TfrmPedido.btnDetailDiscardClick(Sender: TObject);
begin
  If DSDetail.State in [dsEdit,dsInsert] then begin
    If DSDetail.State = dsInsert then begin
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat -
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
      DSDetail.DataSet.Cancel;
    end else begin
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat -
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
      DSDetail.DataSet.Cancel;
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat +
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
    end;
  end else begin
    if TBDEDataSet(DSDetail.DataSet).UpdateStatus = usInserted then begin
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat -
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
      TBDEDataSet(DSDetail.DataSet).RevertRecord;
    end else if TBDEDataSet(DSDetail.DataSet).UpdateStatus = usDeleted
    then begin
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat +
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
      TBDEDataSet(DSDetail.DataSet).RevertRecord;
    end else if TBDEDataSet(DSDetail.DataSet).UpdateStatus = usModified
    then begin
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat -
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
      TBDEDataSet(DSDetail.DataSet).RevertRecord;
      DSMain.DataSet.FieldByName('PED_VALOR').AsFloat:=
        DSMain.DataSet.FieldByName('PED_VALOR').asfloat +
        DSDetail.DataSet.FieldByName('ITE_VALOR').asfloat;
    end;
  end;
end;
end;
end;

```

Regras de Negócio

Além de inserir os registros do pedido e de seus itens, devemos implementar algumas regras de negócio em torno dessas inserções. Como exemplo iremos implementar duas regras de negócio: a verificação do limite de crédito do cliente em compras a prazo e a baixa do estoque do produto.

Essas regras de negócio devem ser implementadas dentro da transação para que sejam efetivadas juntamente com a inserção dos registros. Para simplificar o processo iremos fazer os eventos apenas para inserção do pedido e dos itens. Não iremos tratar portanto a atualização e exclusão de itens ou do pedido.

Vamos implementar duas procedures dentro do *DataModule*.

```

public
  { Public declarations }
  procedure VerificaLimiteCreditoCliente(Cliente: integer; Valor: double);
  procedure UpdateEstoque(Produto: integer; Quant: integer);
end;

procedure TDMSistVendas.VerificaLimiteCreditoCliente(Cliente: integer; Valor:
double);
begin
  QGeral.close;
  QGeral.SQL.Text:='update cliente set cli_debito = cli_debito + :p1 ' +
    'where pes_cd = :p2';
  QGeral.ParamByName('p1').AsFloat:=Valor;
  QGeral.ParamByName('p2').AsInteger:=Cliente;
  QGeral.ExecSQL;
  QGeral.SQL.Text:='select cli_limitecredito - cli_debito from cliente ' +
    'where pes_cd = :p1';
  QGeral.ParamByName('p1').AsInteger:=Cliente;
  QGeral.Open;
  if QGeral.Fields[0].AsFloat < 0 then begin
    ShowMessage('Limite de Crédito do Cliente insuficiente para a compra');
    QGeral.Close;
    Abort;
  end;
  QGeral.Close;
end;

```

Esse método incrementa o débito do cliente e verifica se ele ultrapassou o limite. Caso isso aconteça uma mensagem é mostrada ao usuário e uma exceção é levantada. A alteração deve ser feita primeiro para que o registro se mantenha travado e outro usuário não altere seus valores.

O evento de baixa no estoque é bem similar e é mostrado abaixo.

```

procedure TDMSistVendas.UpdateEstoque(Produto: integer; Quant: integer);
begin
  QGeral.close;
  QGeral.SQL.Text:='update produto set pro_estoque = pro_estoque - :p1 ' +
    'where pro_cd = :p2';
  QGeral.ParamByName('p1').AsFloat:=Quant;
  QGeral.ParamByName('p2').AsInteger:=Produto;
  QGeral.ExecSQL;
  QGeral.SQL.Text:='select pro_estoque from produto ' +
    'where pro_cd = :p1';
  QGeral.ParamByName('p1').AsInteger:=Produto;
  QGeral.Open;
  if QGeral.Fields[0].AsFloat < 0 then begin
    ShowMessage('Estoque insuficiente do Produto: ' + inttostr(Produto));
    QGeral.Close;
    Abort;
  end;
  QGeral.Close;
end;

```

Devemos então, acrescentar as chamadas desses métodos nos eventos de atualização:

```

procedure TDMSistVendas.QPedidoUpdateRecord(DataSet: TDataSet;
UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  If UpdateKind = ukInsert then begin

```

```

        If QPedidoPED_TIPO.NewValue = 'P' then
            VerificaLimiteCreditoCliente(QPedidoPES_CD_CLI.NewValue,
                QPedidoPED_VALOR.NewValue);
        end;
        USPedido.Apply(UpdateKind);
        try
            QItem.ApplyUpdates;
        finally
            QItem.CommitUpdates;
        end;
        UpdateAction:=uaApplied;
    end;

procedure TDMSistVendas.QItemUpdateRecord(DataSet: TDataSet;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    If UpdateKind = ukInsert then begin
        QItemPED_CD.NewValue:=QPedidoPED_CD.NewValue;
        UpdateEstoque(QItemPRO_CD.NewValue, QItemITE_QUANT.NewValue );
    end;
    USItem.Apply(UpdateKind);
    UpdateAction:=uaSkip;
end;

```