Capítulo 4: Usando o Componente TDatabase

Controlando Conexões de Banco de Dados

Todas as aplicações de banco de dados no Delphi usam um componente TDatabase para encapsular a conexão com um banco de dados. Isto ocorre quando você coloca ou não um componente TDatabase em sua aplicação. Se você não colocar um, o Delphi cria um componente TDatabase temporário para você. Contudo, na maioria das aplicações Client/Server você criará explicitamente os seus próprios componentes TDatabase. Instanciando explicitamente um componente TDatabase, você pode criar uma conexão *persistente* de banco de dados.

Se você estiver acessando seus dados atualmente através de um alias BDE, você pode questionar quais as vantagens de usar um componente TDatabase. O componente TDatabase não é necessário para a conexão com um banco de dados, embora o seu uso fornecerá mais controle sobre a conexão. Uma razão para que você use um componente TDatabase ao invés de um alias BDE é realmente apenas uma maneira de encapsulação. Atualmente, se precisarmos mover uma aplicação inteira, o arquivo de configuração IDAPI tem que ser modificado. Se podemos de alguma forma definir os aliases internamente na aplicação, em uma eventual realocação da aplicação, não temos que reconfigurar o arquivo de configuração IDAPI. Também, eliminando o alias no BDE Administrator, fica mais fácil distribuir aplicações pois você não tem que se preocupar com os arquivos de configuração IDAPI existentes.

Em aplicações de banco de dados, sugerimos a criação de um novo DataModule para conter um componente TDatabase. O TDatabase deve ser isolado no seu próprio DataModule porque, um DataModule que inclua um componente Tdatabase, aumenta a complexidade na hora de criarmos descendentes do mesmo.

O componente Database pode ser manipulado através do uso do Database Component Editor.

O Editor aparece como este quando você der um duplo clique sobre o componente.

Existem várias definições que são importantes na configuração dos seus componentes TDatabase. A primeira é o Nome (Name) do Database. Neste exemplo, o chamaremos de “OrderEntry” uma vez que estamos acessando o banco de dados de exemplo OrderEntry.GDB. Depois Driver Name, o qual, neste caso, é o INTRBASE. Quando você seleciona um driver, o botão Defaulds pode ser pressionado para carregar os parâmetros padrões para aquele driver na janela Parameter Overrides. Você então pode editar as entradas existentes ou adicionar novas. Todos os parâmetros existentes em um alias BDE estão disponíveis nas definições de parâmetro de um componente TDatabase.

Em um ambiente Client/Server, um usuário normalmente tem que se conectar a um servidor de banco de dados para acessar os dados. Podemos sobrepor este login prompt desmarcando o checkbox Login Prompt e então definindo o USER NAME e PASSWORD na janela Parameter Overrides. Uma vez que isto sobrepõe a segurança, este método não é recomendado para produção de aplicações, mas é útil para testes. Se você não estiver interessado em manter a segurança, esta pode ser uma opção atraente.

Existem vários outros parâmetros que você pode definir nesta janela. O parâmetro mais importante é Server Name, o qual será o servidor e o caminho para o banco de dados no qual você poderá se conectar. As outras definições serão discutidas mais tarde neste curso.

Quando estiver usando um banco de dados no seu próprio Data Module, você precisará mover o DataModule para o topo na lista de forms Auto-Created para que o mesmo seja criado antes de qualquer form que precise acessar este banco de dados.

Caixas de Diálogo de Login Personalizadas

Se você tiver tentado fazer o upsize de uma aplicação, você terá notado que o servidor requer que o usuário se conecte toda vez que um form, que acesse uma tabela do servidor, for aberto. Os seus usuários ficarão rapidamente cansados. Como discutimos antes, você pode se tornar um herói se remover este problema criando um componente TDatabase explicitamente. Para cada servidor de banco de dados que você precise se conectar, coloque um Database com seu próprio DataModule na sua aplicação. Isto diminuirá os logins necessários dos seus usuários para um, por banco de dados que estiver sendo acessado.

Quando o checkbox LoginPropt, para o componente database, estiver habilitado, o Delphi irá fornecer ao usuário uma caixa de diálogo padrão de Login, que requisitará o nome do usuário e a senha. Se esta caixa de diálogo padrão de Login não for satisfatória para você, uma caixa de diálogo de Login personalizada pode ser criada. Primeiro, você precisará criar um form de caixa de diálogo de Login.

Então, no manipulador de evento OnLogin do database, a caixa de diálogo pode ser exibida para carregar as informações necessárias para a conexão do banco de dados.

```
Procedure TdtmdlOrderEntry.dtbsOrderEntryLogin(
  Database: TDatabase; LoginParams: TString);
begin
  frmLogin := TfrmLogin.create( Application );
  with frmLogin do
  begin
    try
      if ShowModal - mrOK then
      begin
        LoginParams.Values[‘USER NAME’] :=
           edtUserName.Text;
        LoginParams.Values[‘PASSWORD’] :=
           edtPassword.Text;
      end
      else
      begin
        Application.Terminate;
      end; //if
    finally
      Free;
    end; //finally
  end; //with
end;
```

No código acima, nós simplesmente exibimos o form Login personalizado, modalmente. Se o botão OK estiver selecionado, os valores digitados pelo usuário dentro dos Edits são passados para os parâmetros de login do database. OnLogin recebe uma cópia da propriedade Params do componente database (na propriedade LoginParams). A propriedade Params contém todos os parâmetros de conexão de banco de dados para o BDE. Nós então usamos a propriedade Values para definir ou mudar os parâmetros de login.

Se o botão Cancel estiver selecionado, nós simplesmente encerramos a aplicação. Uma solução mais complexa pode incluir algum tipo de looping para garantir ao usuário várias tentativas no caso dele digitar uma senha errada. Nesta solução, o usuário que digitar o login/senha errado é capaz de acessar a aplicação sem qualquer acesso aos dados.

SchemaCaching

Quando uma aplicação de banco de dados abre uma tabela, informações do dicionário de dados sobre a tabela são armazenados na memória. Estas informações incluiem os nomes dos campos, tipos de campos e índices da tabela. Para melhorar a performance, podemos especificar que o BDE armazene as informações destas tabelas localmente. Para fazer isto, simplesmente defina o parâmetro ENABLE SCHEMA CACHE como True.

Entretanto, existem algumas ramificações quando fazemos isto. Se houverem quaisquer mudanças nas tabelas dentro do banco de dados, elas podem causar erros. Por exemplo, se você freqüentemente adiciona colunas a uma tabela, ou adiciona e remove índices de uma tabela, ou muda constraints de validação nos campos, você deve sempre definir Enable Schema Cache como False. As tabelas devem ser estáveis!

Se você utilizar Enable Schema Cache, você deve definir também o parâmetro Schema Cache Dir para um diretório no qual você queira armazenar o cache local. Se a informação de schema cache for alterada depois que sua aplicação tiver sido executada, você pode limpar a informação de cache existente com o método FlushSchemaCache do componente TDatabase. Este método requer que você o chame passando como parâmetro o nome da tabela que deseja atualizar.

Handle

Se você precisa de alguma funcionalidade não encontrada em quaisquer dos componentes da VCL, você pode optar por fazer chamadas diretamente a algumas das muitas funções da API do BDE. As chamadas à API do BDE não devem ser feitas, a menos que a VCL não suporte, através de métodos e propriedades, a funcionalidade que você necessite.

Muitas das funções da API do BDE requerem informações adicionais, além do handle de banco de dados, e algumas retornam tipos de dados complexos. Criar e manipular estas estruturas pode ser tedioso, mas elas poderão oferecer a você o único mecanismo que forneça a informação de que precisa sobre o seu banco de dados.

Temporary

A propriedade Temporary pode ser usada se você estiver lidando com o componente TDatabase criado implicitamente. Uma vez que os componentes TDatabase são criados automaticamente, sempre que um alias definido na configuração do BDE for usado, abri e fechar tabelas dentro da aplicação podem fazer com que o banco de dados disconecte-se e reconecte-se. Se você optar por usar um alias definido dentro da configuração atual do BDE, você pode definir a propriedade Temporary do componente TDatabase alocado inamicamente para False. Isto evitará que a conexão com o banco de dados seja fechada quando não houver tabelas/cursores abertos.

Nota: esta metodologia pode ser perigosa. Definir esta propriedade requer que a aplicação derrube a conexão manualmente quando a mesma for encerrada.

CloseDataSets e DataSets

Se você precisar fechar todos os datasets atualmente abertos dentro de uma conexão de banco de dados, este método os fechará através de um ciclo por meio da propriedade DataSets, que fechará cada dataset sem derrubar a conexão de banco de dados.

Este método difere do método Close uma vez que mantém a conexão de banco de dado e não requer que o usuário conecte-se no servidor quando um novo dataset for aberto.

TraceFlags

Como o componente TSession, o componente TDatabase tem uma propriedade para controlar as opções de trace do SQL Monitor programaticamente. Isto pode ser de grande ajuda no teste de conexões e de performance de SQL para uma determinada conexão de banco de dados. Uma vez que o componente TDatabase existe em um nível mais granular que o componente TSession, os desenvolvedores podem definir uma opção de trace personalizada para cada uma das conexões de banco de dados dentro de sua aplicação.

Isto permite aos desenvolvedores, trabalhando em uma aplicação, ver apenas as instruções SQL de uma conexão ao invés de todas, quando estiverem depurando ou realizando ajustes de performance na mesma. Veja o gráfico no capítulo sobre componentes TSession para ver as diferentes opções para o SQL Monitor.

Transações e Níveis de Isolação de Transações

Na maioria das aplicações de banco de dados, o controle de transação é necessário. Uma transação representa uma única tarefa composta de uma ou mais operações. Estas operações devem ser confirmadas (comittes) ou desfeitas (rolled back) como uma única tarefa, geralmente mantendo as alterações no banco de dados invisíveis a outras transações até este momento. No Delphi, o controle de transação é fornecido por todo acesso a banco de dados, mesmo com tabelas locais, tais como tabelas Paradox.

Por exemplo, uma aplicação pode apagar um pedido de uma tabela de pedidos. Em uma situação do mundo real, isto pode tornar-se uma operação complexa. Para apagar o pedido, os itens associados com este pedido também deverão ser apagados da tabela Itens de Pedidos. A exclusão deste pedido também deverá refletir em uma tabela de Estoque. Se qualquer comissão for paga a um vendedor por este pedido, ela deverá ser reiniciada e o saldo do cliente também precisará ser atualizado. Se em qualquer momento desta seqüência uma operação não puder ser completada, a transação inteira deverá ser desfeita. Com o controle de transação, podemos garantir que todas estas operações tanto serão confirmadas ao mesmo tempo como desfeitas se necessário. Também, usando os níveis de isolamento de transação, podemos determinar como as mudanças em uma transação são vistas pelas outras transações no banco de dados.

Níveis de isolamento de transação, como o próprio nome sugere, controlam o quanto cada uma das transações dos usuários, gerenciadas pelo BDE, são isoladas de outras. Por exemplo, supondo que sua aplicação, no meio de uma transação, precisa de valores armazenados em um campo para completar um cálculo. Ela deve usar o valor das últimas alterações daquele campo ou o valor armazenado no campo quando a transação teve início? Ambos podem ser corretos - dependendo da situação.

A propriedade Transisolation contida no componente TDatabase é usada para editar a decisão. Existem três configurações possíveis para esta propriedade: tiDirtyRead, tiReadCommited e tiRepeatableRead. Baseado em como esta propriedade é definida, o nível (no qual a transação vê o trabalho realizado em outras transações) é determinado. Este tópico será discutido com maior detalhe em um próximo capítulo.

Capítulo 5: Performance em Ambientes Client/Server

Tables vs. Queries

Quando trabalhamos em um ambiente client/server, questões de performance são a maior preocupação. Com tabelas locais, performance não é uma coisa crítica uma vez que geralmente apenas um usuário está tentando acessar o banco de dados. Entretanto, tipicamente, vários usuários tentam acessar bancos de dados baseados em um servidor. Isto pode causar graves problemas de performance. Eis porque otimização é algo tão crítico em aplicações client/server. Otimizar uma aplicação para realizar exatamente as mesmas tarefas com o mínimo de esforço de um banco de dados pode melhorar a performance de uma aplicação de uma maneira impressionante. No capítulo de SQL Monitor, aprendemos como usar o SQL Monitor para ver as chamadas que estavam sendo feitas ao banco de dados que usávamos. Agora, usaremos este conhecimento para determinar qual é a melhor técnica para otimizar aplicações client/server.

Para começarmos, iremos discutir questões de performance baseadas em TTables e TQueries. É importante notar que quando se usa tabelas locais, o componente TTable é geralmente mais rápido. Isto tem haver com como o BDE implementa o código SQL. Entretanto, só porque o componente TTable fornece uma performance melhor em tabelas locais, isto não necessariamente significa o mesmo para tabelas baseadas em servidor. Como veremos neste exemplo, é totalmente o oposto. Usando componente

TQuery em aplicações client/server, você ultrapassará substancialmente a performance de uma aplicação que use componente TTable. Para o exemplo, nós iremos usar a aplicação de exemplo, Query Props. Nós também iremos usar o SQL Monitor para ver porque os componentes Tquery são eficientes que os companheiros, TTable.

Antes de começarmos, existe um speedbutton importante que precisamos mencionar. O speedbutton Toggle, localizado à direita do TDBNavigator, alterna os datasets entre Ativo e Inativo e vice-versa. Usaremos este speedbutton freqüentemente uma vez que a maioria dos recursos demonstrados na aplicação definirão os datasets automaticamente como Inativos.

Vamos falar sobre o que esta aplicação está tentando nos ensinar. Primeiro, há um DBGrid que está exibindo os registros de Customer do banco de dados Interbase OrderEntry.GDB. Existem dois datasets, TTable e TQuery, na aplicação que serão usados para acessar os dados na tabela Customer. Há também um componente TRadioGroup com duas opções, Table e Query. Para essa sessão, usaremos estas opções para ver a diferença entre os componente TTable e TQuery em aplicações client/server. Também, há um groupbox Update Mode, o qual será discutido em detalhe mais a frente neste capítulo. Finalmente, existem dois checkboxs, Filtered e Live Result, e um edit box rotulado Filter Criteria. Todos estes recursos serão discutidos mais adiante nessa sessão.

Para agora, iremos nos concentrar nas opções Table e Query. Antes de executar a aplicação, precisamos abrir o SQL Monitor selecionando o comando de menu **Database | SQL Monitor**.

Agora, queremos ter certeza que o SQL Monitor sempre estará a frente de qualquer aplicação. Isto é feito simplesmente selecionando o speedbutton Always On Top:

Agora estamos prontos para executar a aplicação. Novamente, estamos preocupados apenas com as questões de performance relacionadas com os componente TTable e TQuery. Uma vez que a definição padrão na aplicação é a opção Table, o DBGrid será populado usando um componente TTable.

Uma vez que a aplicação estiver sendo executada, vamos dar uma olhada no SQL Monitor:

Como você pode ver, usando um componente TTable é necessária a execução de seis instruções SQL para o banco de dados Intervase popular o BDGrid. Agora vamos ver quantas instruções SQL são necessárias usando um componente TQuery. Para fazer isto, queremos ver quantas instruções são necessárias quando o banco de dados for conectado inicialmente. Logo precisamos ter a aplicação Query Props inicializada com o dataset Query. Isto é facilmente realizado definindo a propriedade ItemIndex do rdgrpDataset como 1.

Agora execute a aplicação com o SQL Monitor e compare os resultados:

Note que quando usamos o componente TQuery apenas duas instruções SQL foram necessárias para popular o DBGrid. Isto é uma redução significante das instruções SQL

executadas. Esta redução melhora significativamente a performance não só da aplicação, mas também do servidor.

Usar componente TTable em aplicações client/server tem outros problemas além dos que estamos discutindo. Outro problema é que o componente TTable também seleciona todas as colunas e linhas de uma tabela. Se a aplicação só precisa de uma ou duas colunas, usar o componente TTable não será algo muito eficiente. Usando o componente TQuery, carregar todas as linhas e colunas pode ser algo um pouco tedioso no começo, entretanto, os benefícios compensam os problemas.

Outro problema no uso de TTable em aplicações client/server é que os metadados, para as tabelas de banco de dados, são carregados. Os metadados incluem todas as colunas, constraints, definições de domain, etc. associados com a tabela. Geralmente esta informação só deve ser chamada quando necessário. Este é outro problema de sobrecarga envolvendo o uso do componente TTable em ambientes client/server.

Live vs. Não-Live Result Sets

Até aqui, aprendemos que utilizar componentes TQuery oferecem uma maior performance em ambientes cliente/servidor. Agora falaremos sobre como os registros são obtidos. O componente TQuery possui uma propriedade Booleana chamada RequestLive. RequestLive especifica se uma aplicação espera receber um live result set do Borland Database Engine (BDE) quando a query for executada. Essencialmente, isto significa que se RequestLive for False os registros são obtidos da tabela como read-only. Assim, se os usuários quiserem poder atualizar, inserir ou apagar registros, a propriedade RequestLive deve ser configurada como True. Entretanto, como veremos, configurar RequestLive para True causa um extremo overhead, o que por sua vez, reduz a performance.

Para demonstrar a propriedade RequestLive, utilizaremos a aplicação Query Props novamente. A primeira coisa a ser feita é configurar o botão de rádio DatasSet para o componente Query. Em seguida, marcaremos o checkbox Live Result. Lembre-se de ter o SQL Monitor aberto para que possamos visualizar as chamadas sendo feitas ao servidor.

Observe que foi necessário executar cinco instruções para que o resultado fosse atualizável. Se você se recorda, a aplicação necessita somente de duas instruções para popular o BDGrid com dados não atualizáveis. Vamos testar isto com uma tabela. Muito embora um componente TTable não possua uma propriedade RequestLive, ele possui uma propriedade ReadOnly. A propriedade Read Only controla se os usuários podem atualizar, inserir ou apagar dados da tabela. Vejamos o que acontece quando a propriedade ReadOnly está configurada para True.

Para fazer isto, simplesmente marque a caixa Live Result.

Observe que utilizar dados atualizáveis com um componente TTable causou muito menos overhead. Entretanto, ainda é melhor utilizar um componente TQuery. Como veremos no capítulo sobre Cached Updates, podemos resolver este problema da utilização de componente TQuery com dados não atualizáveis utilizando um componente UpdateSQL.

Filtrando

Filtro lida com a “filtragem” de um conjunto de registros de forma que somente dados específicos sejam visualizados. Um exemplo de filtragem seria exibir somente os clientes de um determinado país. Um ótimo recurso de filtragem é que o critério pode ser configurado em tempo de execução. Por exemplo, o usuário poderia especificar de qual país seriam exibidos os clientes. Assim, a pergunta principal seria, como filtramos datasets?

Vamos começar declarando que tanto o componente TTable com TQuery podem ser filtrados da mesma forma. Ambos os componentes possuem propriedades Filtered e Filter que são utilizados para configurar o critério de filtragem. A propriedade Filtered é uma propriedade Boolean que determina se Filter está ativa. A propriedade Filter é uma string que realmente faz a filtragem. No exemplo que discutimos acima, poderíamos configurar a propriedade Filter para “Nation = ‘UNITED KINGDOM’“ e configurar a propriedade Filtered para True. Assim, somente os registros com o valores de Nation igual a UNITED KINGDOM seriam visualizados.

Para testar isto, iremos utilizar a aplicação Query Props. Na aplicação, existe um checkbox Filtered e um edit Filter Criteria. Utilizaremos isto para testar como os componentes TTable e TQuery manipulam a filtragem.

Para começar, vamos testar o componente TTable. Iremos filtrar todos os registros que tenham o valor de Nation igual a UNITED KINGDOM.

Novamente, utilizando o SQL Monitor, execute a aplicação configurada da seguinte maneira:

Primeiro, note que o Filter Criteria requer que somente os registros com o valor Nation igual a UNITED KINGDOM sejam exibidos.

Agora, vejamos o SQL Monitor e o que exatamente ocorreu:

Se examinarmos a janela SQL Text Window, uma nova cláusula SQL foi exibida. A última linha da instrução SQL possui agora uma cláusula Where, Where (Nation = ?). O critério de filtragem preenche o ‘?’ com o valor colocado no edit Filter Criteria.

Assim o TTable exibe somente os registros que não foram filtrados. Embora isto pareça resolver alguns problemas de performance com TTable, ainda há mais benefícios na utilização de componente TQuery em um ambiente cliente/servidor.

Vejamos como filtrar utilizando o componente TQuery. Como o componente TTable, o componente TQuery possui uma propriedade Filtered e uma propriedade Filter. Estas propriedades funcionam da mesma forma para ambos os componentes.

Para ver como estas propriedades afetam a performance, utilizaremos a aplicação Query Props em conjunto com o SQL Monitor.

Novamente, observe que configuramos Filter Criteria em tempo de execução e os únicos registros visualizados são aqueles que se encaixam nos critérios. O que você imagina que ocorreu para obtermos estes registros? Se fizermos uma analogia com o que o componente TTable fez, sua resposta seria que o Delphi adicionou uma cláusula Where à string SQL.

Observe que nada foi adicionado à string SQL. Isto significa que toda a filtragem foi feita localmente. A princípio, esta funcionalidade pode parecer um ótimo recurso; entretanto, examinemos mais de perto. Se uma tabela possui cinco milhões de registros e a propriedade Filter indica que somente dez destes registros serão retornados, o TQuery retornará todos os cinco milhões quando somente dez serão necessários.

Assim, qual a solução? Nem o TTable nem o TQuery possuem muita força quando utilizam as propriedades Filtered e Filter em ambientes cliente/servidor. A solução seria utilizar Parâmetros. Componentes TQuery possuem uma propriedade chamada Params que pode ser utilizada como um dispositivo de filtragem. Podemos especificar um parâmetro na string SQL que essencialmente filtrará somente os registros que queremos visualizar. A aplicação de exemplo a seguir demonstra a utilização de parâmetros para filtragem.

Utilizando Parâmetros

Como aprendemos na última seção, utilizar as propriedades Filtered e Filter em um ambiente cliente/servidor oferece poucos benefícios. Entretanto, podemos utilizar a propriedade Params em queries para resolver este dilema. A aplicação de exemplo a seguir, QueryParams, retorna um conjunto de registros que o usuário define. O usuário simplesmente digita o país desejado na caixa Nation, dá um clique no botão Filter e o TDBGrid exibe os registros para o país selecionado.

Observe que GERMANY foi digitado na caixa Nation e todos os registros foram filtrados com exceção daqueles cujo valor de Nation é GERMANY.

Em nossa string SQL, estamos configurando a cláusula Where para obtermos somente os registros desejados (Where Nation = :NationID). O “dois pontos” antes de “NationID” significa que NationID é um parâmetro. Se olharmos a caixa de diálogo Params veremos NationID listado como um parâmetro:

Data deve ser configurado como string, uma vez que o campo Nation é do tipo string. O campo Value poderia ser deixado em branco, entretanto, demos o parâmetro NationID um valor inicial de ALGERIA. Este valor inicial assegura que a aplicação será iniciada com o DBGrid populado com somente alguns registros. Agora vamos utilizar todas as opções em Trace Options no SQL Monitor. Estamos fazendo isto para ver se todos os registros estão sendo retornados quando configuramos nosso parâmetro.

Para testar isto, limpe o conteúdo do SQL Monitor e digite UNITED KINGDOM na caixa Nation para ver quantos registros são retornados:

Se visualizarmos o SQL Monitor, podemos ver que somente cinco registros foram retornados pelo banco de dados. A filtragem foi feita no servidor e somente os registros necessários foram retornados. Não utilizando as propriedades Filtered e Filter para

“filtrar” registros indesejáveis, aumentamos bastante a performance de nossa aplicação cliente/servidor.

SQL Update Modes

O último tópico neste capítulo é UpdateMode. UpdateMode é uma propriedade enumerada que simplesmente especifica como o Delphi deve encontrar registros a serem atualizados. Existem três configurações disponíveis para atualização. Estas três configurações são as seguintes: upWhereAll, upWhereChanged e upWhereKeyOnly.

upWhereAll

A opção default para a propriedade UpDateMode é upWhereAll. A configuração upWhereAll atualiza um registro somente se todos os campos no registro coincidirem exatamente com os valores especificados na cláusula Where. Em termos de uma aplicação cliente/servidor, quando dois usuários estão tentando atualizar diferentes campos no mesmo registro ao mesmo tempo, somente um usuário poderá fazer a atualização. O outro usuário não conseguirá encontrar o registro que deseja atualizar. Como você pode ver, isto possui um enorme grau de segurança uma vez que somente um usuário poderá atualizar um registro por vez.

Vejamos como utilizar a configuração upWhereAll afeta uma aplicação cliente/servidor. Utilizaremos a aplicação Query Props novamente em conjunto com o SQL Monitor.

Vamos executar a aplicação, configure DataSet para Query e marque a caixa Live Result:

Agora limpe o SQL Monitor e mude o campo Address no DBGrid. Atualiza o registro e veja no SQL Monitor o que ocorreu:

Observe a cláusula Where na instrução SQL. Todos os campos do registro foram comparados. Se um dos campos fosse alterado, a instrução Where não poderia encontrar o registro a ser atualizado. A mesma coisa aconteceria se tivéssemos utilizando a opção Table com UpdateMode configurado para o upWhereAll.

UpWhereChanged

Na última seção, aprendemos sobre a configuração upWhereAll para a propriedade UpdateMode. Agora vejamos a configuração upWhereChanged. A configuração upWhereChanged atualiza um registro baseado em colunas chave e colunas modificadas. Para demonstrar isto, executaremos a aplicação Query Props novamente com DataSet configurado para Query e Update Mode configurado para Where Changed. Faça qualquer alteração no campo Addres e atualize o registro.

Observe que os únicos campos necessários na clausula Where foram Customer_Key e Addres. Contanto que o campo chave não seja alterado, outros usuários podem fazer quaisquer alterações em outros campos no registro. Embora esta configuração não

forneça muita segurança, é muito mais eficiente do que utilizar a configuração upWhereAll. Novamente, o mesmo se aplica ao componente TTable.

upWhereKeyOnly

A última configuração disponível na propriedade UpdateMode é upWhereKeyOnly. A configuração upWhereKeyOnly atualiza um registro baseado em colunas chaves específicas. Como veremos, esta é, de longe, a configuração mais eficiente; entretanto, esta configuração deve ser utilizada somente se uma aplicação tiver acesso exclusivo ao dado.

Vejamos novamente a aplicação Query Props com SQL Monitor. Configure DataSet para Query e Update Mode para Where Key Only. Marque também a caixa Live Result. Agora faça uma alteração no campo Addres. Se necessário, antes de utilizar o registro, limpe o SQL Monitor.

Desta vez, o único campo necessário foi a chave primária Customer-Key. Esta configuração é de longe a menos restritiva das três configurações Como em muitas situações com computadores, desistindo de uma coisa você ganha em outro lugar. Neste caso, estamos desistindo completamente da segurança para ganhar performance.

Capítulo 6:Processamento de Transação Cliente / Servidor

O que veremos neste capítulo:

- A diferença entre controle de transação Implícita e Explícita.
- Manipulação de transação utilizando o componente TDatabase.
- Cached Updates.
- Transisiolation Levels.
- SKLPassThruMode.

Gerenciando Transações de Banco de Dados

No Delphi, as transações são controladas *implicitamente* por default e elas são controladas pelo BDE. Bem, o que isto significa? Controle implícito de transação significa que uma transação separada é utilizada para cada registro gravado no banco de dados. Cada transação é gravada ou cancelada para cada registro individual.

Utilizando controle implícito de transação, você minimiza conflitos multi-usuário para acesso de dados e suas aplicações. A razão para isto é não termos múltiplos registros amarrados a uma única transação o que em alguns casos pode levar diversos minutos para ser completado. Isto também fornece uma visualização mais consistente do banco de dados a seus usuários. Entretanto, em controle implícito de transação, o tráfego na rede aumenta porque cada registro precisa ser escrito imediatamente no banco de dados. Isto também pode tornar a sua aplicação mais lenta.

Para gerenciar suas transações mais eficientemente, você pode *explicitamente* ou controlar manualmente as transações em suas aplicações em banco de dados. Controle explícito de transação significa que é você que seleciona quando iniciar, efetuar ou cancelar transações. Existem duas formas nas quais você controla explicitamente transações em uma aplicação. A primeira, é utilizando as propriedades e métodos do componente TDatabase. A segunda é utilizando passthrough SQL com o componente TQuery. Passthrough SQL utiliza SQL Links para passar instruções SQL ao servidor do banco de dados.

Existem três métodos primários a serem utilizados para manipular transações através do componente TDatabase. O método *StartTransaction* faz exatamente o que o nome diz: inicia uma transação. Quando é chamado, todas as gravações subsequentes no banco de dados são parte desta transação. A transação pode ser efetuada (commit) ou cancelada (rollback) utilizando os métodos *Commit* ou *Rollback* do TDatabase.

Nota: É uma boa prática manipular transações com estes métodos em um bloco try...except. Se você tentar efetuar a transação neste bloco e ocorrer um erro, você pode tentar manipular o erro e tentar o commit novamente, ou cancelar a transação no código de manipulação de exceção.

Por exemplo, vamos criar uma nova aplicação utilizando o componente TDatabase OrderEntry em seu próprio DataModule.

Crie um novo form Master/Detail utilizando o Form Wizard:

O próximo passo nos solicita a tabela a ser utilizada como tabela mestre deste form. Neste exemplo, iremos ver quais itens de inventário um fornecedor fornece. Assim utilizaremos a tabela Supplier:

Agora devemos selecionar quais campos queremos utilizar de nossa tabela mestre. Deixemos o campo Comment de lado.

Quando der um clique no botão Next, o Form Wizard solicita uma segunda tabela para a parte do detalhe deste form.

Como mencionado, utilizaremos a tabela Inventory como nosso detalhe:

Novamente, deixaremos o campo Comment de lado:

Selecionando um estilo Grid para a parte de detalhe do form, o usuário pode ver diversos itens do inventário associados a um fornecedor de uma vez.

Ele também auxilia a distinguir visualmente os dois tipos de informação:

Em seguida, devemos permitir que a aplicação saiba como ligar as duas tabelas. Selecione o campo Supplier_Key em ambas as caixas de lista e dê um clique no botão Add:

O join será exibido na caixa de lista da parte inferior. O passo final envolve a geração do form. Iremos utilizar este form como nosso Main Form.

Iremos também criar um form com um DataModule:

Após o Form Wizard terminar de criar o form, iremos modificá-lo para incorporar o processamento de transação. A transação deve ser iniciada no evento BeforeEdit com o método StartTransaction. Isto irá armazenar todas as alterações do banco de dados até que o método Commit seja utilizado, gravando todas as alterações no banco de dados, ou o método Rollback seja utilizado para cancelar as modificações pendentes.

O form Supplier utiliza um relacionamento um-para-muitos entre as tabelas Supplier e Inventory. Estas tabelas são parte do DataModule gerado pelo Form Wizard. Desta forma, o código a seguir será adicionado ao evento BeforeEdit da tabela mestre (Supplier):

```
procedure TDtMdlSupplier.TblSupplierBeforeEdit(DataSet:
  TDataSet);
begin
  dtmdlOrderEntry.dtbsOrderEntry.StartTransaction;
end.;
```

Esta instrução inicia uma transação no banco de dados. Agora, qualquer alteração feita no banco de dados é parte desta transação. Se o usuário escolher salvar as alterações feitas neste form utilizando o botão post na barra de navegação, precisará terminar a transação através do método Commit do banco de dados.

O código a seguir é acionado ao evento AfterPost para conseguirmos isto:

```
procedure TDtMdlSupplier.TblSupplierAfterPost(DataSet:
  TDataSet);
begin
  dtmdOrderEntry.dtbsOrderEntry.Commit;
end;
```

Se o usuário escolher cancelar quaisquer alterações feitas, precisaremos terminar a transação utilizando uma instrução rollback. O código a seguir cancelará quaisquer transações e atualizará a tabela Inventory. A atualização é necessária porque a instrução rollback não atualiza automaticamente os valores alterados. Em tabelas, o usuário pode utilizar o método Refresh. Entretanto, com queries não existe um método Refresh. O usuário terá que utilizar outras técnicas para atualizar a exibição de dados em uma query. Este tópico será discutido no capítulo 7 sobre atualização de queries.

```
Procedure TDtMdlSupplierAfterCancel(Dataset:
  Tdataset);
begin
  dtmdlOrderEntry.dtbsOrderEntry.Rollback;
  tblInventory.Refresh;
end;
```

Agora, digamos que o usuário selecione o botão Edit e inicie uma transação. Então, imagine que ele tenta fechar o form sem efetuar ou cancelar a transação. Para evitar que o form seja fechado até que seja feito um post (commit) ou cancel (rollback), o

usuário deve adicionar o código a seguir no manipulador de evento OnCloseQuery do form.

```
Procedure TfrmSupplier.FormCloseQuery(Sender: Tobject;
  var CanClose: Boolean);
begin
  if dtmdlOrderEntry.dtbsOrderEntry.InTransaction Then
  begin
     ShowMessage('Cannot close the form while the
       transaction is active.  Please Rollback or Commit
       your transaction');
     CanClose := False;
  end;
end;
```

Neste código, utiliza-se a propriedade In Transaction do componente TDatabase. Esta é uma propriedade Booleana que indica se uma transação de banco de dados está em progresso ou não. Neste exemplo, se uma transação estiver em progresso, é exibida uma mensagem ao usuário indicando o problema e configurada CanClose para False para interromper o fechamento do form.

Cached Updates

Cached updates são para datasets o que transações explícitas são para bancos de dados. Eles permitem que o usuário modifique múltiplos registros dentro de uma tabela e mantenha estas alterações em um buffer. Assim, para o usuário final, o resultado final é o mesmo de quando o processamento da transação é utilizado. Cached Updates funcionam armazenando as alterações em um buffer, localmente. Quando Cached Updates são habilitados e o usuário insere um registro no modo de edição, uma cópia dos dados é feita no registro (ou registros) e as alterações são aplicadas à cópia local. Quando o usuário estiver pronto para gravar as alterações, o Delphi verifica se o registro no banco de dados não foi alterado e então aplica as alterações. Se o registro foi alterado, uma exceção de banco de dados é gerada, indicando que o registro foi alterado desde a última "verificação". Pode-se usar o componente Update SQL (discutido em um capítulo posterior) para forçar as atualizações nesta situação. Uma discussão e exemplo de Cached Updates serão mostrados em capítulos futuros.

Transisolation Levels

Transisolation Levels, como o nome implica, controla a extensão na qual cada transação dos usuários é isolada das outras. Por exemplo, suponha que uma aplicação, no meio de uma transação, precise do valor armazenado em um campo para completar o cálculo. O valor deve ser obtido das alterações mais recentes feitas no campo ou do valor armazenado no campo quando a transação teve início? Ambos podem estar corretos, dependendo da situação.

TiDirtyRead

A propriedade Transisolation contida no componente TDatabase é utilizada para ditar a decisão. Uma configuração, tiDirtyRead, deve ser utilizada se você quiser os dados mais atuais, não importando se foram efetuados no banco de dados ou não. Neste caso, sua aplicação irá consultar continuamente o banco de dados à procura de atualizações feitas por outros usuários, mesmo que ainda não estejam gravadas. O perigo em potencial aqui é que sua transação pode utilizar dados que um usuário resolva cancelar. A maioria dos servidores de banco de dados não suporta dirty reads.

TiReadCommitted

Para recuperar somente dados que tenham sido gravados, o usuário deve utilizar a configuração tiReadCommitted. Os dados mais recentes ainda serão utilizados, mas somente após terem sido gravados. Esta configuração é preferível a dirty reads, especialmente porque a maioria dos servidores a suporta.

TiRepeatableRead

A última opção para esta prioridade é tiRepeatableRead. Ela isola completamente uma transação das alterações de outros usuários. Ela também força uma transação a utilizar os mesmos valores durante a transação inteira. Esta configuração pode causar problemas ambientes de transações intensivas, uma vez que ela mantém todas as transações completamente isoladas das outras. Um cenário possível poderia ser uma aplicação de Estoque/Vendas. Suponha que duas pessoas precisem vender um item e há somente um no estoque. Ambos poderiam vender a peça e decrementar o estoque para zero. Neste caso, o cliente de uma destas pessoas ficaria bastante desapontado, sem mencionar que a quantidade no inventário não estaria correta. Para cenários assim, não é recomendado utilizar tiRepeatableRead com frequencia. Alem disso, poucos servidores suportam este tipo de acesso a dados.

Níveis não suportados

Diferentes servidores de banco de dados podem ou não suportar diferentes níveis de transisolation. Por exemplo, se uma aplicação configurar Transisolation para tiDirtyRead, que a maioria dos servidores não suporta, o BDE utilizará o próximo nível mais alto suportado pelo servidor. A tabela a seguir resume os níveis de isolamento suportados pelos servidores reconhecidos pelo BDE:

Servidor	**Nível Especificado**	**Nível Real**	
Oracle	tiDirtyRead		
		tiReadCommitted	
		tiRepeatableRead	tiReadCommitted
		tiReadCommitted	

		tiRepeatableRead	(READONLY)
Sybases, MS-SQL	tiDirtyRead		
		tiReadCommitted	
		tiRepeatableRead	tiReadCommitted
		tiReadCommitted	
		Not supported	
DB2	tiDirtyRead		
		tiReadCommitted	
		tiRepeatableRead	tiDirtyRead
		tiReadCommitted	
		tiRepeatableRead	
Informix	tiDirtyRead		
		tiReadCommitted	
		tiRepeatableRead	tiDirtyRead
		tiReadCommitted	
		tiRepeatableRead	
InterBase	tiDirtyRead		
		tiReadCommitted	
		tiRepeatableRead	tiReadCommitted
		TiReadCommitted	
		tiRepeatableRead	
Paradox, dBase, Access, FoxPro			
		tiReadCommitted	
		tiRepeatableRead	tiDirtyRead
		Not supported	
		Not Supported	

SQLPassThruMode

SQLPassThruMode é outro parâmetro contido pelo componente TDatabase que utiliza tabelas baseadas em servidores. Ele controla como as transações são manipuladas dentro de um banco de dados. Quando estiver utilizando o driver InterBase, o defauts é NOT SHARED. Esta configuração pode fazer com que seus objetos TQuery utilizem uma conexão diferente de outros componentes Delphi (tal como TTables) que tenham acesso a dados. Se estes dois tipos de conexão se misturarem, sua aplicação pode receber resultados não confiáveis na atualização de dados. A configuração mais eficiente na maioria das plataformas é SHARED AUTOCOMMIT. Ela indica que, a cada linha de dados utilizada, SQL e atualizações baseadas em métodos serão gravadas no servidor. Uma terceira configuração para este parâmetro é SHARED NOAUTOCOMMIT. Isto resulta em tanto SQL quanto atualizações baseadas em métodos utilizando a mesma conexão. Entretanto, a aplicação deve controlar manualmente todas as transações. Desta forma, é recomendável utilizar NOT SHARED para se assegurar que SQL e atualizações baseadas em métodos não conflitem.

Capítulo 7: Atualizando Queries

O que veremos neste capítulo:

- Problemas com inserção de registros com queries.
- Resolvendo problemas de inserção através da atualização de queries.
- Problemas com atualização de queries.
- Solução para atualização de queries.

Agora estamos prontos para adicionar nosso componente TDatabase. Como nos exemplos de capítulos anteriores, utilizaremos novamente o componente TDatabase OrderEntry em seu próprio DataModule. A propriedade RequestLive do componente TQuery deve estar configurada para True e a propriedade SQL deve conter algo parecido com o seguinte:

```
Select Customer_Key,  Name,  Address,  Nation,  Phone,
      Account_Balance
From Customer
Order By Customer_Key
```

Uma vez que todos os componentes de dados tenham sido condectados, nosso form deve ter um grid que exiba os dados de clientes.

Agora vamos executar nossa aplicação e inserir um novo registro, Customer#151:

Agora gravaremos o registro e visualizaremos o final do grid para ver se o novo registro foi inserido:

Observe que parece que o novo registro não foi inserido em na tabela. Entretanto, para todos os efeitos, o registro foi inserido. O problema é que a query precisa ser atualizada.
A solução mais lógica para isto seria simplesmente dar um clique no botão Refresh no DBNavigator.

A razão para esta exceção é que o botão Refresh do DBNavigator está tentando chamar o método Refresh do dataset no qual está conectado. Entretanto, diferente das tabelas, queries não possuem um método Refresh. Assim, como atualizar facilmente a query de forma que qualquer inserção seja refletida nos controles data-aware?

Para resolver isto, coloca-se um speedbutton no painel. Elimina-se também o botão Refresh do DBNavigator, configurando nbRefresh para False na propriedade VisibleButtons. Isto é necessário para que o usuário não se confunda com múltiplos botões Refresh:

Você deve ter observado que, quando a aplicação foi fechada para fazer as alterações, a query foi atualizada. Isso ocorre porque a query foi fechada e reaberta neste ponto. Isto é o que faremos no código para atualizar a query.

Assim, no evento OnClick do speedbutton, fecharemos e reabriremos a query depois, para que os dados apresentem as alterações. O código segue abaixo:

```
procedure TfrmQueryExample.SpeedButton1Click(Sender:
```

```
  Tobject);
begin
  with gryCustomer do
  begin
    DisableControls;
    Close;
    Open;
    EnableControls;
  end;
end;
```

A primeira coisa a ser notada no código é uma chamada ao método DisableControls. DisableControls evita que os dados sejam exibidos em controles data-aware. Ele essencialmente desabilita os controles data-aware de forma que, quando fechamos a query, os dados não desaparecem. Agora a query pode ser fechada e reaberta sem que o usuário veja o grid piscar. Após a query ter sido reaberta, o método EnableControls é utilizado para permitir que os dados sejam exibidos nos controles data-aware.

Vamos executar a aplicação novamente e adicional um novo registro, Customer#152. Vá ao final do grid para verificar se a query foi atualizada:

Agora dê um clique no botão Refresh para ver que o novo registro foi adicionado:

Algumas coisas devem ser observadas sobre o botão Refresh: primeiro, o speedbutton fez a atualização de forma que o grid agora reflete o que realmente há na tabela. Segundo, quando utilizamos o botão Refresh, fomos colocados novamente no primeiro registro da tabela. Isto ocorre porque, quando uma query é aberta, ela assume que você quer ver o início da tabela. Isto pode ser um grande problema particularmente quando estiver utilizando tabelas com milhares ou milhões de registros. Seria fácil um usuário se perder ou simplesmente se irritar sempre que atualizasse a query. A seção seguinte demonstra como resolver esse tipo de problema.

Topo do Result Set

Como demonstrado na última seção, queries atualizáveis podem apresentar um problema interessante. Quando uma query é fechada e depois reaberta, você sempre será posicionado no topo da tabela. À primeira vista, isto pode não parecer muito problemático; entretanto, isto pode causar muitos problemas se o usuário inserir registro frequentemente.

Para resolver este problema, utilizaremos o método Locate. O método Locate pesquisa um dataset a procura de um registro específico e, caso seja localizado, ele posiciona o cursor nele. Este método retorna um valor Booleano que pode ser utilizado pelo programador para diferenciar se Locate encontrou o registro. Se Locate retornar False o registro atual não é alertado. Geralmente você irá querer utilizar um campo que tenha valores únicos associados a ele. Dois tipos de campos que podem ser utilizados são a chave primária e chaves candidatas. Uma chave primária é um campo único que distingue registros. Chaves candidatas são campos ou grupos de campos que são únicos. Um bom exemplo de um campo chave candidata é Social Security Number. Social Security Numbers não são utilizados geralmente como chaves primárias, mas

normalmente são únicas. Para nosso exemplo, não temos um bom “candidato” para uma chave candidata. Ao invés, utilizaremos os campos Name e Phone em combinação como chave candidata.

Para começar, devemos atribuir a duas variáveis os valores dos campos Name e Phone no qual o usuário esteja quando ele ou ela de um clique no botão Refresh. Utilizaremos NameMarker e PhoneMarker como estas variáveis.

```
var
   NameMarker, PhoneMarker : String;
```

Agora devemos atribuis às variáveis os valores atuais dos campos Name e Phone na query. É importante fazer esta atribuição antes de fecharmos e abrirmos a query. Se fizermos a atribuição após a query ter sido reaberta, os valores dos campos serão incorretos.

```
Procedure TfrmQueryExample.spdbtnRefreshClick(Sender:
   Tobject);
var
  NameMarker, PhoneMarker : String;
begin
  with gryCustomer do
  begin
     NameMarker := FieldByName('Name').AsString;
     PhoneMarker := FieldByName('Phone').AsString;
     DisableControls;
     Close;
     Open;
     EnableControls;
     Locate('Name;Phone", VarArrayOf([NameMaker,
        PhoneMarker]), [loCaseInsensitive]);
  end;
end;
```

Agora estamos prontos para utilizar o método Locate. O primeiro parâmetro pelo qual precisamos passar são os campos chave. Os campos chave (em nosso caso, campos de chave candidata) pelos quais iremos passar são Name e Phone. Os parâmetros seguintes são os valores reais das chaves. Atribuímos estes valores às variáveis NameMarker e PhoneMarker. Finalmente, o método Locate nos permite definir determinadas opções na localização de registros. As opções disponíveis são loCaseInsensitive e loParcialKey. A opção loCaseInsensitive simplesmente procura por campos chave e valores chave sem se preocupar com maiúsculas ou minúsculas. A opção loParcialKey permite que os valores chave incluam apenas parte da chave a ser localizada. É importante observar que estas poções de localização são valores do tipo set e devem ser colocadas entre colchetes.

Agora estamos prontos para executar a aplicação. Para testá-la, iremos inserir um novo registro, Customer#153, gravar o registro e imediatamente dar um clique no botão Refresh:

Note: agora que a query foi atualizada, Customer#153 está visível no DBGrid e fomos posicionados no novo registro.