# Construindo Aplicações com ClientDataSet e InterBase Express

Isto é um documento técnico de uma conversa ocorrida na 12ª conferencia anual dos desenvolvedores Borland
Por Bill Todd – The Database Group, Inc.

*Bill Todd é presidente da Database Group, Inc.Empresa de consultoria em Banco de dados e Desenvolvimento que fica perto de Phoenix. Ele é co-autor de quatro livros de programação de Banco de dados e mais de 80 artigos, e é um membro do time que dá suporte técnico no newsgroup da Borland.Ele é um orador freqüente nas conferencias de desenvolvedores Borland nos EUA e Europa. Bill também é um conhecido professor e tem diversas classes de programação nos EUA e outros paises.*

Introdução
Problemas com Interbase Express
Usando ClientDataSet
Construindo o Data Module
Construindo a interface do usuário
Alterando os Dados
Tratando Erros de Atualização
Ordenando os Dados
Mostrando o Status do Registro

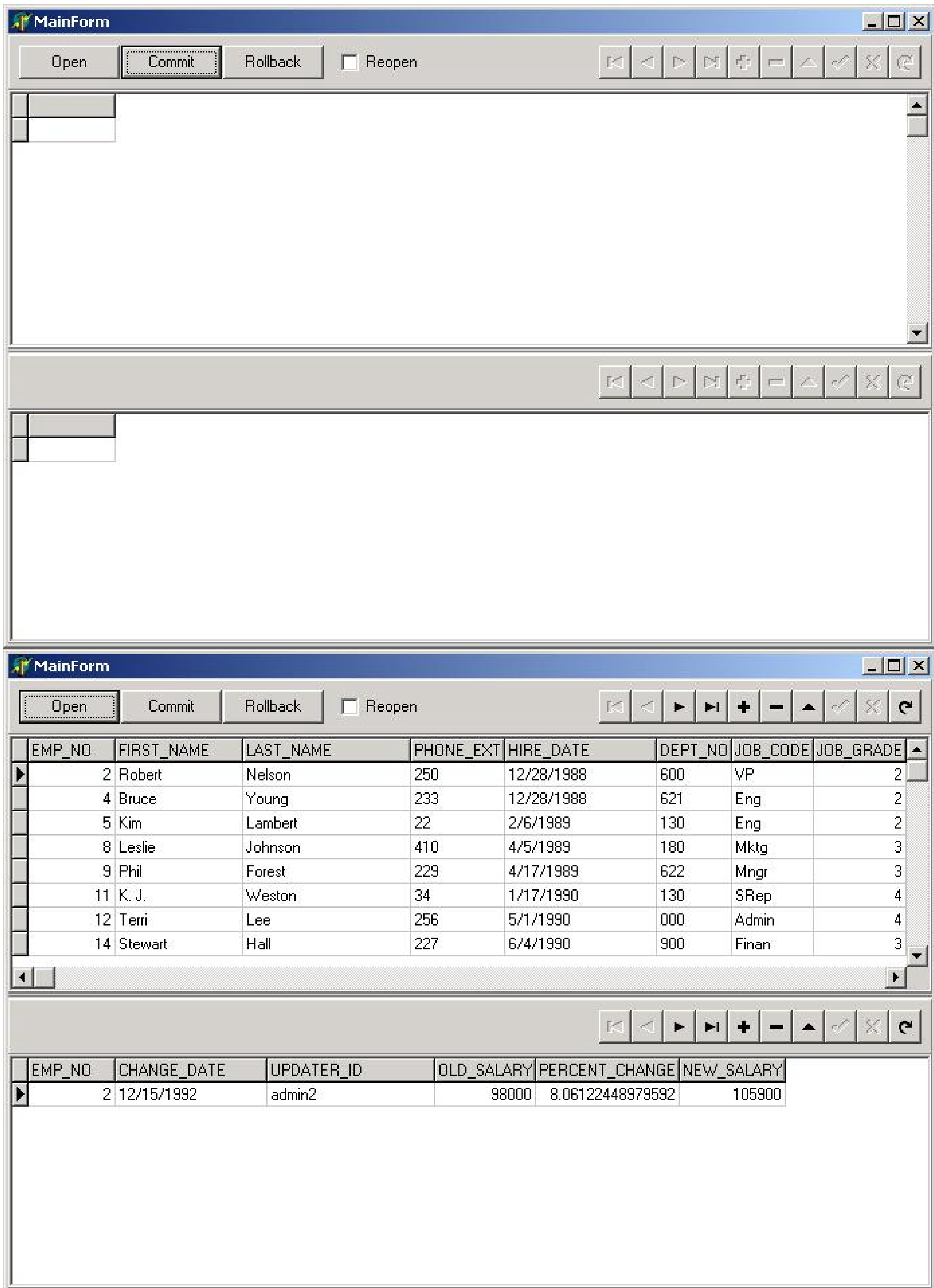
---

## Introdução

Agora que o componente ClientDataSet está disponível nas versões Professional e EnterPrise do Delphi e C++ Builder, é hora de olhar as vantagens de usar os ClientDataSets em combinação com os componentes Interbase Express para construir aplicações Interbase. Como você verá neste texto existem muitas vantagens em usar ClientDataSets em suas aplicações. Esta é a razão porque o componente ClientDataSet é a base da nova arquitetura Dbexpress da Borland.
Existem também muitas razões pra usar Interbase Express (IBX) para construir aplicações Interbase. Primeiro, o IBX é muito menor e fácil de desenvolver que o BDE. Segundo, o IBX tem acesso mais rápido ao banco de dados Interbase que o BDE. Finalmente, IBX dá acesso a características do Interbase que não estão disponíveis pelo BDE tais como múltiplas transações simultâneas, grande variedade de níveis de isolação das transações e acesso a CommitRetaining e RollbackRetaining.

## Problemas com Interbase Express

O maior problema que desenvolvedores encontram trabalhando com os componentes IBX é que todo acesso a um banco de dados Interbase deve ser feito dentro do contexto de uma transação. Isto significa que você não pode sequer ler ou mostrar o conteúdo de um registro sem iniciar uma transação. O problema é que quando você dá commit ou rollback na transação, todos os datasets incluídos no componente IBTransaction são fechados e o usuário de repente esta olhando uma tela que não mostra nenhum registro.
Você pode ver o efeito no programa IBXDemo que acompanha este texto. A tela seguinte mostra o form principal após o botão open ter sido clicado. A próxima tela mostra o form após o click no botão Commit




Lógico que existem modos de lidar com isto, e a aplicação IBXDEMO mostrará um deles. Se você verificar o checkbox Reopen antes de clicar no botão Commit, o programa salva o código da chave primaria das tabelas employee e salary history, confirma (commit) a transação, reabre os dois IBDataSets e posiciona nos registros que estavam antes do commit. A mesma coisa acontece se você clicar no botão Rollback. Isto funciona, mas é um bocado de trabalho extra.

Outra solução é seguir a filosofia normal do propósito Cliente/Servidor que é iniciar o usuário com uma tela em branco e pedir um critério de seleção para trazer um pequeno numero de registros para trabalho. O usuário pode então fazer quaisquer alterações nos registros e confirmar(commit) as alterações. Quando as alterações são confirmadas o usuário estará novamente com uma tela em branco para selecionar outro conjunto de registros. Este acesso requer menos código, mas pode deixar as transações abertas por um tempo longo demais. Por exemplo, suponha que um usuário precise alterar centenas de registros dos clientes do estado do Paraná. Se o usuário selecionou todos os clientes do Paraná de uma vez, ele gastará diversas horas de trabalho para fazer tudo. Isso causa dois problemas em potencial. Primeiro, se seu computador travar, todas as alterações serão desfeitas(rollback) e todo o trabalho será perdido. Segundo, a transação ficará aberta por muitas horas com centenas de registros alterados e travados e não poderão ser alterados por outros usuários.
Outra solução que muitos desenvolvedores usam é chamar CommitRetaining ou RollbackRetaining ao invés de Commit ou Rollback. Num primeiro momento isso parece ser a solução ideal, pois não fecha as tabelas e permite ao usuário continuar a ver e alterar os registros selecionados. Entretanto, CommitRetaining e RollbackRetaining não fecham sua transação. Isso significa que você está usando um isolamento de transação do momento e você não pode ver quaisquer alterações feitas por outros usuários até você confirmar (commit) ou desistir (Rollback). Também significa que você desabilitou a "garbage collection" do Interbase forçando ele a manter a transação aberta e as informações dos registros de quando a transação foi iniciada. Isso causa problemas graves de performance em um sistema multiusuário, com muitos usuários alterando diversos registros. Isso é nitidamente o caso em que a cura é pior que a doença.

## Usando ClientDataSet

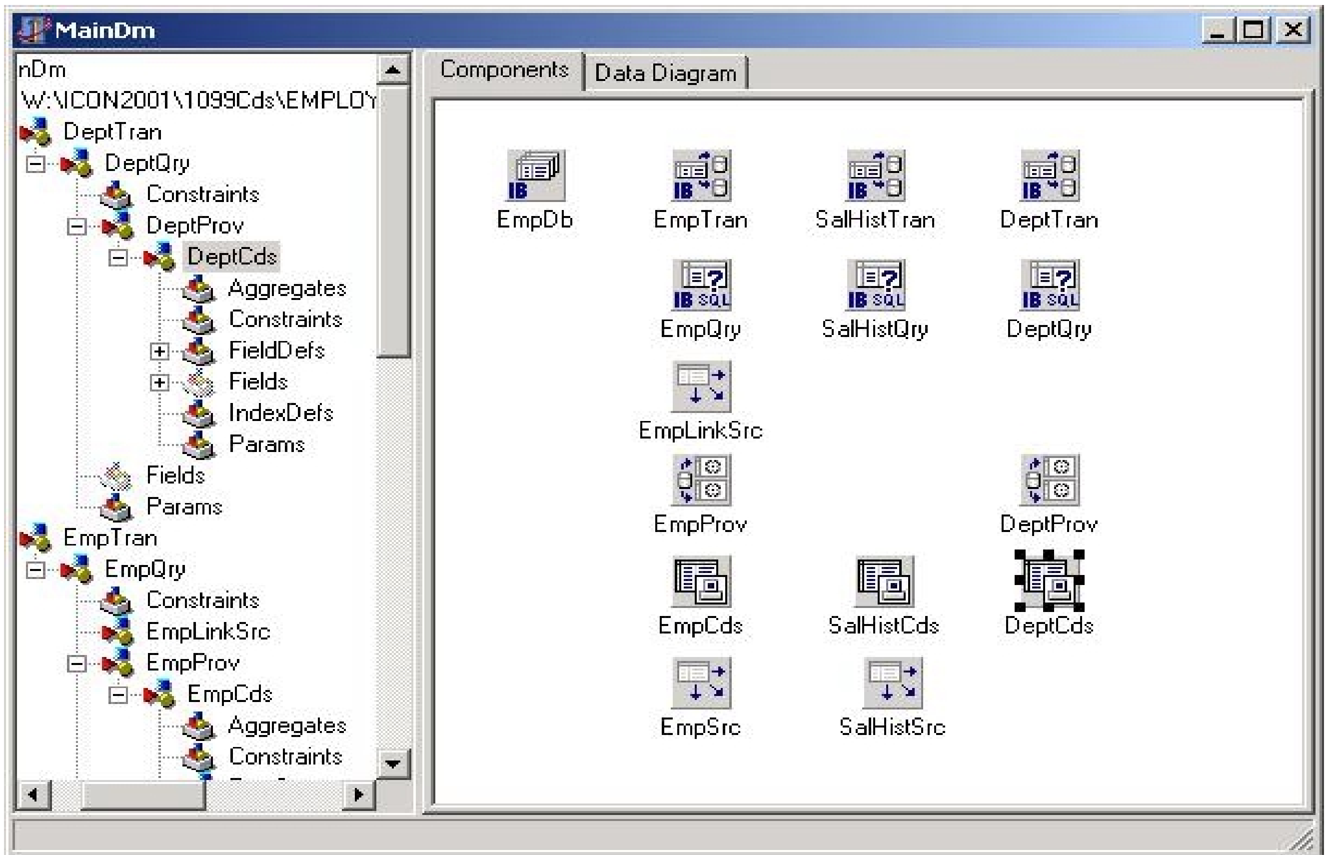
O componente ClientDataSet é um dos componentes da palheta Midas, usado para implementar a estratégia de prover/resolver a edição de registros que foi introduzido como parte do Midas no Delphi 3. Antes de entrar em detalhes de construir uma aplicação que usa os componentes DataSetProvider, ClientDataSet e IBX, vale a pena examinar os fundamentos da estratégia de prover/resolver do Midas.

### A estratégia Prover/Resolver

O mecanismo prover/resolve usa um componente DataSetProvider para enviar os dados de um componente Dataset a um ClientDataSet. O componente ClientDataSet guarda os registros recebidos do provedor na memória. Usuários podem inserir, apagar e alterar os registros e as alterações também são guardadas na memória pelo ClientDataSet. Quando o usuário terminar as alterações, uma chamada ao método ApplyUpdates do ClientDataSet envia as alterações para o provider. O provider então gera as instruções SQL necessárias para executar as alterações no banco de dados (inicia uma transação, envia as instruções SQL para o servidor e, se não houver erros, confirma (commit) a transação).
Numa explicação básica, você pode ver as vantagens desta arquitetura. Primeiro, as transações ficam abertas muito pouco tempo. Quando você abre um ClientDataSet que está conectado a um DataSetProvider, o provider é que abre o dataset que está conectado a ele. Quando o dataset é aberto, ele executa sua instrução de SQL SELECT e retorna os registros para o provedor que então os envia para o ClientDataSet fechando o dataset e confirmando (commit) a transação de leitura. Quando você chama o método ApplyUpdates do ClientDataSet as alterações são enviadas para o provider que gerará as instruções SQL necessárias, inicia a transação, envia as instruções SQL para o banco de dados e confirma (commit) a transação. Isto faz com que as transações de leitura e escrita sejam bem rápidas, o que significa que outros usuários não estarão bloqueados por longo tempo. Transações rápidas também significa que o numero de transações abertas ao mesmo tempo serão menores e o acesso ao banco de dados menor. A segunda grande vantagem desta estratégia é te dar o controle da quantidade de dados a serem trabalhados, uma vez que você controla a instrução SQL que é executada, você controla quantos registros serão trazidos do servidor o que te da o controle sobre o trafego que será gerado na sua rede e o quanto de memória você usará para armazenar os registros. Isto será particularmente valioso se você estiver desenvolvendo uma aplicação que será usada numa WAN ou num link de baixa velocidade.

## Construindo o Data Module

A próxima figura mostra o data module da aplicação de exemplo CDSIBX.




Este data module é projetado para suportar o form que mostra a relação um-para-muitos entre a tabela Employee e a tabela Salary_History. A tabela employee é selecionada pelo departamento. Por outro lado o dataset Employee inclui um campo lookup no campo Dept_No o qual mostra o nome do departamento. O componente IBDatabase no canto superior esquerdo está configurado exatamente como estaria numa aplicação que somente usa componentes IBX.
A seguir vem três componentes IBTransaction, um para cada tabela usada na aplicação. Quando usar ClientDataSets você deve usar um componente de transação separado para cada tabela uma vez que diferentes tabelas podem interagir com o banco de dados simultaneamente. A seguir usa-se um componente IBQuery para cada tabela. Verifique que a propriedade Unidirectional de todos os componentes IBQuery está setado como True para reduzir o consumo de memória e aumentar a performance. Uma vez que estes componentes query serão somente usados para trazer o dados da query para o componente DataSetProvider não há necessidade de retornar os dados. Embora o componente IBQuery traga um conjunto de dados read-only, isso será mais que necessário uma vez que alterações no banco de dados serão controladas automaticamente pelo componente DataSetProvider. A propriedade SQL do componente EmpQry será:

```
Select * from EMPLOYEE Where Dept_No = :Dept_No order by Last_Name, First_Name
```

então os registros de um único departamento serão selecionados. A instrução SQL para o componente SalHistQry é:

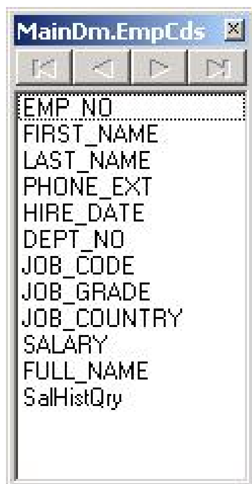
```
select * from SALARY_HISTORY where Emp_No = :Emp_No order by Change_Date desc
```

Neste caso o parâmetro, :Emp_No, tem o mesmo nome que o campo da chave primaria da tabela Employee e a propriedade DataSource do componente SalHistQry esta setada para EmpSrc, o componente DataSource que esta conectado ao componente EmpQry. Isto determina ao componente SalHistQry para pegar o valor do parâmetro do registro corrente de EmpQry. Isto traz somente os

registros do histórico de salários dos empregados que esta selecionado. A propriedade SQL do componente DeptQry e:

```
select DEPT_NO, DEPARTMENT from DEPARTMENT order by Dept_No
```

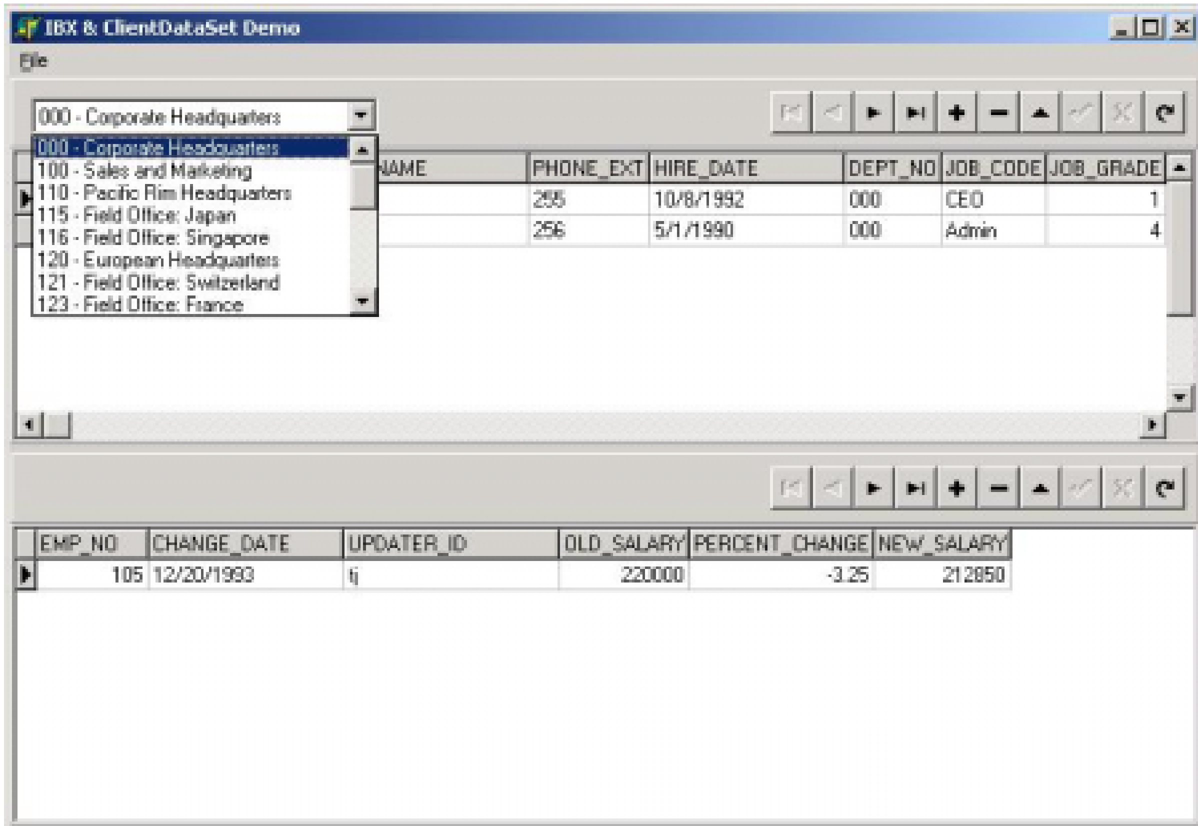
o qual seleciona todos os nomes e números dos departamento.
Um único componente DataSetProvider é o provider do componente EmpQry. A propriedade Dataset do provider esta setada para EmpQry. Nenhum provider é necessário para o componente SalHistQry porque seus registros serão incluídos num campos do dataset nos registros trazidos pelo DataSetProvider do EmpQry. Isto acontece automaticamente para qualquer componente filho que é ligado a um componente pai através da propriedade Datasource. Um segundo componente DataSetProvider é ligado ao componente DeptQry para trazer os registros do departamento.
Finalmente três componentes ClientdataSet serão usados para fornecer os dados para a interface do usuário da aplicação. O primeiro EmpCds é conectado ao provider EmpProv, definindo a propriedade ProviderName do ClientDataSet para EmpProv. Se você está usando ClientDataSet em uma aplicação Multitier verifique se a propriedade RemoteServer será deixada em branco, uma vez que o provider esta na mesma aplicação como o ClientDataSet. Após conectar ao ClientdataSet EmpCds ao seu provider, o próximo passo é abrir o Editor de Campos (fields Editor) e inserir todos os campos como mostrado abaixo.




Observe o ultimo campo SalHistQry. Este é um campo cujo tipo é TdataSetField e ele contem o registro do histórico de salários devolvido pelo dataset SalHistQry. Você **deve** instanciar o campo no editor de campos para o campo ficar disponível, ou você não terá acesso aos registros do histórico de salários. Agora você pode incluir o componente SalHistCds e definir sua propriedade DataSetField para SalHistQry usando o campo drop-down. Isto conecta o Clientdataset SalHistCds ao campo SalHistQry no dataset employee. O terceiro Clientdataset DeptCDs está conectado ao DatasetProvider DeptProv pela propriedade Providername. Finalmente dois componentes Datasource são conectados aos clientdataset EmpCds e SalHistCds.

## Construindo a interface do usuário

Uma vez que estamos interessados nos componentes de acesso aos dados, a interface do usuário no exemplo CDSIBX não é nada especial. A tela principal neste ponto é mostrada abaixo:




O form contem dois grids e dois DBNavigators para mostrar os registros dos empregados e do histórico de salários respectivamente. Para fazer esta aplicação parecer uma cliente/servidor normal, onde o usuário somente trabalha com um subconjunto dos dados, existe um combo box que mostra os departamentos. Selecionando um departamento no combo box, automaticamente seleciona os empregados daquele departamento usando o código no evento OnChange do combo box, como mostrado abaixo.

```
procedure TMainForm.DeptComboChange(Sender: TObject);
begin
  with MainDm do
  begin
    EmpCds.Close;
    EmpCds.Params.ParamByName('Dept_No').AsString :=
      Copy(DeptCombo.Items[DeptCombo.ItemIndex], 1, 3);
    EmpCds.Open;
    SalHistCds.Open;
  end; //with
end;
```

Este código fecha o Clientdataset EmpCds, coloca os três primeiros caracteres do item selecionado no combo box no parâmetro Depto_No do EmpCds e reabre EmpCds. A propriedade Params do Clientdataset é uma lista do objeto TParam e cada objeto TParam representa um dos parâmetros na instrução SQL do componente dataset que o DataSetProvider do ClientDataSet está ligado, neste caso EmpQry. Isso Significa que você pode facilmente alterar o(s) valor(es) da instrução SQL e trazer um novo grupo de registros a qualquer tempo.
Se você quer mais flexibilidade, você pode usar a propriedade CommandText do ClientDataSet para alterar toda a instrução SQL. Suponha que você queira deixar sua aplicação apta a selecionar os empregados tanto por departamento ou por grade de salários. A aplicação Exemplo tem uma opção de

seleção do menu principal que oferece estas duas escolhas. O form realmente contem dois painéis posicionados no topo de cada um. Um tem o combo box para escolha do departamento e o outro tem um Edit Box para permitir ao usuário digitar a grade. Quando a aplicação inicia, o painel do departamento está visível e o painel de grades não. Aqui o código para o evento OnExecute do item EmpByGrade é anexado ao item de menu Employee By Grade.

```
procedure TMainForm.EmpByGradeExecute(Sender: TObject);
begin
  DeptPanel.Visible := False;
  GradePanel.Visible := True;
  with MainDm.EmpCds do
  begin
    Close;
    with MainDm.EmpQry.SQL do
    begin
      Clear;
      Add('SELECT * FROM EMPLOYEE WHERE JOB_GRADE = :JOB_GRADE ORDER BY LAST_NAME,
FIRST_NAME');
    end; //with
    Params.Clear;
    FetchParams;
    GradeEdit.SetFocus;
  end; //with
end;
```

Este evento inicia fazendo o painel do departamento invisível e o painel de grade visível. A seguir ele fecha o Clientdataset EmpCds, limpa a propriedade SQL do IBQuery e insere uma nova instrução na propriedade SQL. Finalmente o código limpa a propriedade Params para remover o parâmetro Dept_No e chama o método FetchParams do ClientDataSet's para criar o parâmetro da nova instrução SQL. Para selecionar os empregados pela grade apenas digite um numero de grade no edit box e tecle Enter. O código seguinte do evento OnKeyDown do TEdit fecha o ClientDataSet, seta a grade ao parâmetro Job_Grade e reabre o ClientDataSet.

```
procedure TMainForm.GradeEditKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  Grade:      Integer;
begin
  if Key = VK_RETURN then
  begin
    with MainDm.EmpCds do
    begin
      try
        Grade := StrToInt(GradeEdit.Text);
      except
        Grade := 0;
      end; //try
      Close;
      Params.ParamByName('Job_Grade').AsInteger := Grade;
      Open;
      MainDm.SalHistCds.Open;
    end; //with
  end; //if
end;
```

### Alterando os dados

Quando você insere, apaga e altera registros, suas alterações são armazenadas na propriedade Delta do ClientDataSet. Para enviar estas alterações para o banco de dados você deve chamar o método ApplyUpdates do ClientDataSet. Até você chamar o evento apply updates você pode desfazer as alterações em ordem reversa, desfazer todas as alterações do registro corrente ou desfazer todas as alterações de todos os registros. O menu Edit na aplicação de Exemplo dará todas estas escolhas. O código do evento OnExecute dos itens associados a cada item do menu é mostrada abaixo.

```
procedure TMainForm.UndoCurrentRecordExecute(Sender: TObject);
{Desfaz todas as alterações no registro corrente.}
begin
  MainDm.EmpCds.RevertRecord;
end;

procedure TMainForm.UndoLastChangeExecute(Sender: TObject);
{Desfaz a ultima alteração e move-se p/ o registro alterado.}
begin
  MainDm.EmpCds.UndoLastChange(True);
end;

procedure TMainForm.CancelChangesExecute(Sender: TObject);
{Cancela todas as alterações.}
begin
  MainDm.EmpCds.CancelUpdates;
end;

procedure TMainForm.ApplyUpdatesExecute(Sender: TObject);
{Se existirem alterações não executadas na propriedade Delta então execute-as.}
begin
  with MainDm.EmpCds do
  begin
    if ChangeCount > 0 then
    begin
      ApplyUpdates(-1);
      Refresh;
    end; //if
  end; //with
end;
```

O método UndoLastChange, mostrado no terceiro evento acima, pega um parâmetro boleano. Se o parâmetro é true a ultima alteração será desfeita e o registro se tornará o registro corrente a menos que exista um registro inserido mais novo. Se o parâmetro é false a ultima alteração é desfeita, mas o registro corrente não muda, a menos que a ultima alteração tenha feito o registro corrente desaparecer por ser uma inserção e ter sido a ultima alteração.

O ultimo evento acima chama o método ApplyUpdates. Antes de chamar o método ApplyUpdates o código verifica se a propriedade ChangeCount é maior que zero. ChangeCount retorna o numero de alterações não confirmadas na propriedade Delta. Se ChangeCount for zero, é porque não existem alterações a serem confirmadas então não há razão para executar o método ApplyUpdates.

ApplyUpdates usa um único parâmetro que é o numero de erros permitidos antes de abortar o processo. Suponha que existem dez registros esperando para serem confirmados e você chama o método ApplyUpdates com o parâmetro MaxErros igual a um. Se o segundo registro causar um erro o processo irá parar e não haverá meios de tentar confirmar as outras oito alterações no banco de dados. Se o parâmetros MaxErrors for setado para menos um não haverá limites para o numero de erros permitidos. Toda alteração tentará ser confirmada mesmo quer todas causem erros.

ApplyUpdates previne alterações em registro que foram alteradas por outros usuários desde o momento em que você leu o registros incluindo a clausula WHERE em todas as instruções de INSERT, DELETE e UPDATE geradas pelo DataSetProvider. A propriedade UpdateMode do controle DataSetProvider cujos campos foram incluídos na clausula WHERE. UpdateMode pode ser setada para:

1. **upWhereAll** (default) - Se UpdateMode for setada para upWhereAll todos os campos que não forem BLOB do registro serão incluídos na clausula WHERE. Isto significa que você obterá um erro quando você confirmar suas alterações se outro usuário alterou qualquer campos do registro mesmo se o campo que o outro alterou não seja o campo que você alterou.
2. **upWhereChanged** - Usando upWhereChanged inclui somente os campos que você alterou na clausula WHERE, então você somente obterá um erro se outro usuário alterou os campos que você está alterando. Esta é a configuração que provavelmente você usará na maioria dos casos.
3. **upWhereKeyOnly** - A opção upWhereKeyOnly inclui somente os campos da chave primária (primary key) na clausula WHERE. Isto significa que se outro usuário alterar um campo e você alterar o

mesmo campo, suas alterações sobreporão as alterações do outro usuário. Existe porem uma limitação. A clausula WHERE nunca inclui campos BLOB, então se outro usuário alterar o registro desde o memento que você o leu e somente alterou o campo BLOB, você não será avisado quando atualizar o registro.

**Alterando Datasets que não podem ser alteradas**

Trabalhar com registro retornados por uma stored procedure no servidor tem um principal problema. Você não pode atualizar os registros. Todavia, você pode alterar Datasets de uma stored procedure quando você usa ClientDataSet e DataSetProvider. Isto é fácil porque o dataSet Provider já gera as instruções SQL para executar as atualizações. O único problema é os registros vem de uma Stored Procedure e o provider não tem meios de achar o nome da tabela que deveria ser atualizada.
Felizmente, o DataSetProvider tem um evento OnGetTableName que dispara quando gerado a expressão SQL para solicitar atualizações. Tudo o que você tem a fazer é escrever uma linha de event handler que determina o nome da tabela para atualizar para os eventos de parâmetro TableName e você poderá agora solicitar atualizações para registros retornados por uma a stored procedure.
Essa técnica é também útil para instruções SQL SELECT que poderiam não ser normalmente atualizáveis. Suponha que você tenha uma multi-table join query que seja necessário ao usuário poder ver informações de duas ou mais tabelas, mas o usuário só precisa fazer mudanças para a área de uma tabela. Tudo o que precisa fazer é instanciar os objetos do campo no tempo planejado para o ClientDataSet e usar a propriedade ReadOnly para todos os campos que o usuário não pode mudar. Então cria-se um evento OnGetTableName para o DataSetProvider fornecer o nome da tabela em que as atualizações deveriam ser aplicadas.
A vida não é tão fácil se você tem uma join multi-table e usuários precisam mudar registros em duas ou mais tabelas. Nesse caso você vai precisar escrever um evento BeforeUpdateRecord para o componente DataSetProvider A propriedade Delta do ClientDataSet é transmitida para esse evento em propriedade DeltaDs O tipo de propriedade do DeltaDs é TClientDataSet então, você precisa acessar a todas as propriedades e métodos do componente ClientDataSet em seu evento. Nesse evento você precisará transferir os registros da propriedade do DataSet e gerar suas próprias instruções SQL para aplicar as atualizações. Você pode usar o método UpdateKind para descobrir se o processo de registro pode ser inserido, apagado ou modificado. Os objetos de campo do parâmetro DelltaDs têm um OldValue e uma propriedade NewValue. Isso permite a você usar DeltaDs.FieldByName('SomeField').OldValue para obter o velho valor do campo e DeltaDs.FieldByName('SomeField').NewValue para obter um novo valor de campo. Obviamente registros para serem deletados têm somente velhos valores e registros a serem inseridos têm somente novos valores. Registros modificados são cuidados de uma forma não intuitiva, mas faz sentido uma vez que você o compreende. Para cada registro que é modificado existem dois registros adjacentes no DataSet. O primeiro registro tem um status atualizado para usUnmodified e é o registro como foi dito anteriormente, nenhuma mudança foi feita . O segundo registro é de status atualizado para usModified e contém valores para os campos que foram mudados apenas. As propriedades de OldValue e NewValue dos objetos de campo são variáveis Quando você examina o NewValue para um campo de registro usModified ele será: Unassigned se o valor do campo não for alterado, Null se o valor do campo for mudado para algo sem validade e terá um novo valor se o campo for alterado para nulo ou qualquer valor para outro valor. Você pode usar as funções VasIsEmpty e VarIsNull para ver se um campo foi modificado ou modificado para nulo. Usando essa informação você pode gerar suas próprias instruções SQL INSERT, DELETE e UPDATE para cada tabela que tiver que atualizar, determinar as instruções SQL para propriedades SQL de um componente IBSQL e executa-lo. Enquanto você processa cada registro, altere o parâmetro Applied para True então o provider saberá para não executar as atualizações.

**Tratando erros de atualização**

Quando você chama o ApplyUpdates a propriedade Delta é enviada para o DataSetProvider que gera instruções SQL para aplicar cada modificação para o database e executa-lo. Se um erro ocorrer o registro é salvo. Quando todas as atualizações forem tentadas ou quando a conta de erro máximo permitido for alcançada, ou o que vier primeiro, nenhum erro será retornado para o ClientDataSet que dispara um evento OnReconcileError. Então, você deve fornecer um evento OnReconcileError para distribuir com algum erro que ocorra. Felizmente, engenheiros do Borland fazem isso fácil. Se você

selecionar File | New do menu Delphi e ir para a página de Dialogs do repositório você irá encontrar um form chamada Reconcile Error Dialog. Tudo o que precisa fazer é adicionar essa form para sua aplicação, adicionar essa unit na cláusula uses de cada unit que contém um ClientDataSet e adicionar o código mostrado abaixo para cada evento ClientDataSet's OnReconcileError.

```
procedure TMainDm.EmpCdsReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Quando um erro ocorre enquanto estão sendo aplicadas as atualizações esse código usa uma caixa de diálogo para exibir o que mostra o registro, as modificações feitas no registro, o tipo de atualização tentada (insert, delete or modify) e a mensagem de erro. O usuário pode editar esse registro e optar por uma das seis ações seguintes A ação selecionada é determinada por um evento Action parameter.

1. raSkip – Não aplica atualizações mas deixa as modificações na propriedade Delta.
2. raAbort – Anula toda a operação de atualização. Nenhuma modificação será aplicada.
3. raMerge - Atualiza o registro mesmo que tenha sido alterada por outro usuário.
4. raCorrect – Troca as modificações no Delta com as alterações feitas por outro usuário.
5. raCancel – Desfaz todas as modificações do registro.
6. raRefresh – Desfaz todas as modificações do registro e relê o registro do banco de dados.

Você pode modificar a aparência ou atributo do diálogo de reconciliação de erros do repositório ou escrever seu próprio evento OnReconcileError do zero, que atenda suas necessidades.

### Sorting Data On-the-Fly

O fato de que o ClientDataSet guardar os registros na memória em sua propriedade Data é uma grande vantagem. Um exemplo disso é que você pode ordenar os registros por qualquer campo ou combinação de campos simplesmente designando o nome do campo para a propriedade IndexFieldNames do ClientDataSet. Para ordenar por mais de um campo separe os nomes dos campos com ponto e vírgula. O código seguinte é o evento OnTitleClick para o DBGrind que mostra os registros dos empregados.

```
procedure TMainForm.EmpGridTitleClick(Column: TColumn);
begin
  MainDm.EmpCds.IndexFieldNames := Column.FieldName;
end;
```

Esta única linha do código permite ao usuário clicar no título de qualquer coluna para ordenar os registros empregados dessa coluna. Essa técnica tem uma restrição e duas limitações. A restrição é que você não pode usar IndexFieldNames em um campo calculado do dataset, como os registros do histórico dos salários na aplicação exemplo. De fato, não há maneira de mudar a ordenação de um campo calculado no dataset exceto mudando a cláusula ORDER BY da query. A primeira limitação é que a ordenação é sempre ascendente. Não há maneira de especificar uma ordenação descendente usando propriedades IndexFieldNames. Segundo, se você tiver um grande número de registros no ClientDataSet, por exemplo, 50,000, a ordenação pode levar alguns segundos( ou minutos).
Ambas as limitações podem ser facilmente superadas usando indexes. Você pode criar indexes em um ClientDataSet em tempo de projeto usando a propriedade IndexDefs ou em runtime usando o método AddIndex. Ambos os métodos permitem a você criar indexes ordenados ascendentemente por alguns campos e descendentemente por outros bem como os indexes que são case insensitive em um ou mais campos. Por todos os registros estarem na memória criar um index é muito rápido. Por exemplo, criar um index de 50,000 registros leva só alguns segundos. Criar um index de 1,000 registros é basicamente instantâneo. A aplicação de exemplo inclui um index criado em tempo de projeto que ordena os registros empregados descendentemente quanto pelo salário e ascendentemente pelo nome. A figura seguinte mostra o IndexDef para o index no Object Inspector e o código seguinte a figura do Object Inspector mostra como você pode criar o mesmo index usando o método AddIndex do ClientDataSet.

```
MainDm.EmpCds.AddIndex('DescBySalary', 'Salary;Last_Name;First_Name',
    [ixDescending], 'Salary');
```

Nessa chamada ao AddIndex o primeiro parâmetro é o nome do index, o segundo é a lista do nome dos campos a serem indexados, a terceira é o parâmetro de opções do index e a quarta são os nomes dos campos a serem ordenados em ordem descendente. O método AddIndex do ClientDataSet pode ter dois parâmetros adicionais que não serão usados nesse exemplo. O primeiro é a lista de campos que serão usados em casos insensitivos e o segundo em nível de ordem de grupo. Nível de ordem do grupo é usado como campo agregado para acumular subtotais. Campos agregados serão descritos mais tarde.

O código seguinte é o evento OnExecute da ação DescendingBySalary que é chamada pelo item de meu Descending By Salary.

```
procedure TMainForm.DescendingBySalaryExecute(Sender: TObject);
begin
  if DescendingBySalary.Checked then
  begin
    DescendingBySalary.Checked := False;
    MainDm.EmpCds.IndexFieldNames := gByNameFieldNames;
  end else begin
    DescendingBySalary.Checked := True;
    MainDm.EmpCds.IndexName := gDescBySalaryIndex;
  end; //if
end;
```

Essa ação troca entre o index DescBySalary e a ordenação por nome usando a propriedade IndexFieldNames. O evento OnClick do item Sort1 do menu, mostrado abaixo, verifica o valor corrente da propriedade EmpCds.IndexName e seta a propriedade Checked do item de ação em caso da ordenação ter sido modificada por um click na barra de título do grid.

```
procedure TMainForm.Sort1Click(Sender: TObject);
begin
  if MainDm.EmpCds.IndexName = 'DescBySalary' then
    DescendingBySalary.Checked := True
  else
    DescendingBySalary.Checked := False;
```

**Mostrando o status do registro**

Você pode facilmente mostrar o status de um registro, isso é, se tiver sido modificado, inserido ou não modificado, usando um campo calculado. O campo de status do ClientDataSet Employee é um campo calculado e o código mostrado é um evento OnCalcFields.

```
procedure TMainDm.EmpCdsCalcFields(DataSet: TDataSet);
begin
  case EmpCds.UpdateStatus of
    usModified:     EmpCdsStatus.AsString := 'Modified';
    usUnModified:   EmpCdsStatus.AsString := 'Unmodified';
    usInserted:     EmpCdsStatus.AsString := 'Inserted';
    usDeleted:      EmpCdsStatus.AsString := 'Deleted';
  end;
end;
```

O Chamado ao método UpdateStatus do ClientDataSet retorna a uma das quatro constantes, usModified, usUnmodified, usInserted ou usDeleted. Claro que você nunca verá um registro cujo status é usDeleted.

### Filtrando Dados

Uma maneira do clientdataset ajudar você a reduzir o trafego da rede e a carga do servidor é filtrando os registros em um ClientDataSet ao invés de executar outro query. Por exemplo, suponha um usuário selecione todos os registros do cliente de um estado e você precise ver os registros de somente as vendas de um território naquele estado. Você pode exibir somente os registros de um território em uma das três maneiras. Primeiro, se você tem um edis do campo do território e ele é o index ativo que você pode usar o método SetRange do ClientDataSet mostrado abaixo.

```
CustomerCds.SetRange([23], [23]);
```

Esse método de chamada vai restringir sua visão dos dados para somente aqueles registros no território 23. Para remover a seleção e ver todos os registros chama-se o método CancelRange. A segunda técnica é aplicar um filtro usando a propriedade filter do ClientDataSet. Os filtros ClientDataSet usam sintaxe SQL WHERE e são, então mais poderosos que os filtros BDE que você deve ter usado no passado. O código seguinte mostraria os registros para o território 23 usando um filtro.

```
with CustomerCds do
begin
  Filter := 'Territory = 23';
  Filtered := True;
end;
```

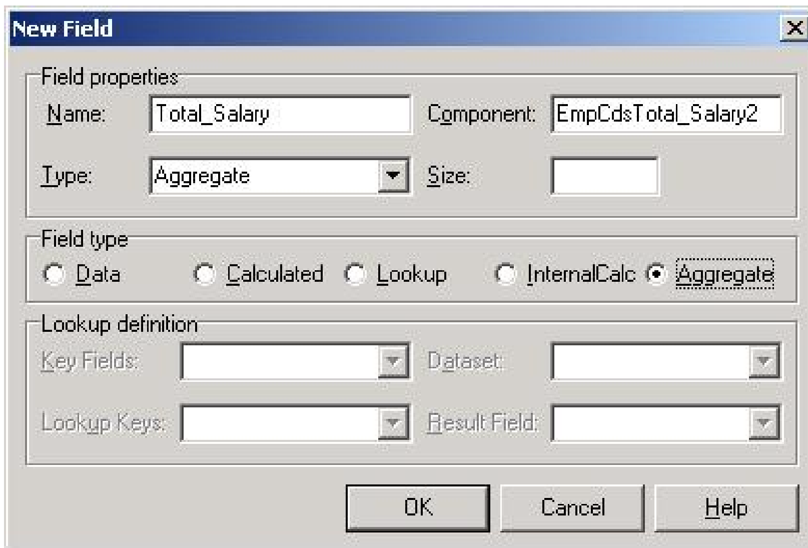
Para remover o filtro coloque a propriedade Filtered para False. A terceira maneira de filtrar registros é escrever um evento OnFilterRecord.

### Usando tabelas de lookup Locais.

Outra forma de reduzir o tráfico de rede e a carga do servidor é usar ClientDataSets para todos lookup tables que você pode usar p/ validação dos dados, Combo Boxes e para suportar campos de lookup no ClientDataSets. Para tabelas com poucos milhares de registros ou menos você tem que selecionar todos os registros de um servidor. Então, enquanto você não chama o método Refresh do ClientDataSets, sua aplicação usará os registros guardados na memória da propriedade Data do ClientDataSets e nunca relerá esses registros do servidor.

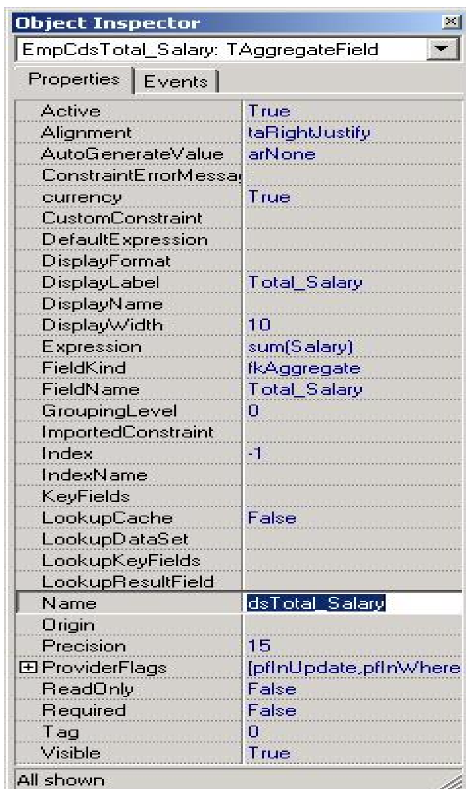
### Campo agregados.

Campo agregados permite que você defina valores de sumario que serão automaticamente mantidos pelo ClientDataSet. Para criar um campo agregado use o Fields Editor para adicionar um novo campo, cujo tipo é aggregate, no ClientDataSet como mostrado abaixo.




Campo agregados são mostrados numa seção separada do Fields Editor abaixo dos outros campos como mostrado abaixo.

Quando selecionado o campo Total_Salary que é um campo agregado, aparecerá como mostrado abaixo no Object Inspector.



Para usar o campo agregado você deve definir a propriedade Active p/ True e digitar uma expressão que descreve o valor que você quer na propriedade Epression. Você deve definir também a propriedade AggregatesActive do ClientDataSet p/ True. Neste exemplo a expressão é uma simples soma (sum(salary)), que calcula o total de salários de todos os registros dos empregados no ClientDataSet. Você pode também usar as funções Min, Max, Avg e Count. E Pode também construir expressões complexas. Por exemplo, o campo SalHistCds do ClientDataSet tem um campo agregado cuja expressão é: sum(New_Salary) - sum(Old_Salary)

para calcular o total dos salários alterados do empregado selecionado. O mesmo resultado pode ser obtido com a expressão Sum(New_Salary - Old_Salary). As duas maiores restrições para as expressões campo agregado são que você não pode combinar funções, então Avg(Sum(New_Salary - Old_Salary)) não é certo e você não pode usar valores de campos non-aggregated na expressão porque não existe meio de determinar de qual registro veio o valor.
Você pode também calcular agregados para grupo de registros. Por exemplo, você pode ter subtotais e medias por grupo. A próxima tela da aplicação exemplo mostra o form que é aberto quando você seleciona o item View| Employee Statistics no menu. Verifique que as três colunas da direita do grid mostram o total de salários, a media dos salários e o numero de empregados por departamento.

Employee Statistics By Department

| FULL_NAME | DEPT_NO | SALARY | Department_Salary | Department_Avg | Dept_Count |
|---|---|---|---|---|---|
| Bender, Oliver H. | 000 | $212,850.00 | $266,643.00 | $133,321.50 | 2 |
| Lee, Terri | 000 | $53,793.00 | $266,643.00 | $133,321.50 | 2 |
| MacDonald, Mary S. | 100 | $111,262.50 | $155,262.50 | $77,631.25 | 2 |
| Yanowski, Michael | 100 | $44,000.00 | $155,262.50 | $77,631.25 | 2 |
| Baldwin, Janet | 110 | $61,637.81 | $130,442.81 | $65,221.41 | 2 |
| Leung, Luke | 110 | $68,805.00 | $130,442.81 | $65,221.41 | 2 |
| Ichida, Yuki | 115 | [illegible],000,000.00 | $13,480,000.00 | $6,740,000.00 | 2 |
| Yamamoto, Takashi | 115 | 7,480,000.00 | $13,480,000.00 | $6,740,000.00 | 2 |
| Bennet, Ann | 120 | $22,935.00 | $95,779.69 | $31,926.56 | 3 |
| Reeves, Roger | 120 | $33,620.63 | $95,779.69 | $31,926.56 | 3 |
| Stansbury, Willie | 120 | $39,224.06 | $95,779.69 | $31,926.56 | 3 |
| Osborne, Pierre | 121 | $110,000.00 | $110,000.00 | $110,000.00 | 1 |
| Glon, Jacques | 123 | $390,500.00 | $390,500.00 | $390,500.00 | 1 |
| Ferrari, Roberto | 125 | 9,000,000.00 | $99,000,000.00 | $99,000,000.00 | 1 |
| Lambert, Kim | 130 | $102,750.00 | $189,042.94 | $94,521.47 | 2 |

O primeiro passo para criar um campo agregado que usa grupos é criar um índice que agrupe os registros que você deseja. Na aplicação exemplo um índice para os campos Dept_No, Last_Name e First_Name foi definido na propriedade IndexDefs do ClientDataset EmpStatsCds. A propriedade IndexName do ClientdataSet foi setada para este índice que foi nomeado AscByDept. Alem disso a propriedade GroupingLevel do índice foi setada para 1 para indicar que somente queremos agrupar pelo primeiro campo do índice. Ele poderia ser setado para um valor maior até o numero de campos do índice. Criar campos agregados que usa grupos é o mesmo que criar um que não usa exceto que você deve definir duas propriedades adicionais. A primeira é a propriedade GroupingLevel do campo agregado que deve ser setado para o numero de campos que você que agrupar. Neste exemplo nos agrupamos pelo primeiro campo do índice que é Depto_No, então GroupingLevel foi definido como um. A segunda propriedade é IndexName. Neste exemplo a propriedade IndexName do campo agregado foi definida com AscByDept. O campo agregado somente será calculado quando este índice for o índice ativo do ClientDataSet.

### Clonando Datasets

Outra característica bastante útil do ClientDataSet é a habilidade de Clonar Cursores. Suponha que você precise ver dois diferentes registros do mesmo Dataset ao mesmo tempo. Se você não está trabalhando com ClientDataSets você tem de procurar o registro e salvar ele num array ou usar um outro componente Query para buscar o registro, o que cria trafego e carrega o servidor. Se você está usando ClientDataSets e precisa de uma visão adicional de seus dados, tudo o que você precisa fazer é incluir outro ClientDataSet na sua aplicação e chamar o método CloneCursor como mostrado no seguinte código no evento OnCreate do form EmpCloneForm na aplicação exemplo.

```
procedure TEmpCloneForm.FormCreate(Sender: TObject);
begin
  with EmpCloneCds do
  begin
    CloneCursor(MainDm.EmpCds, False, False);
    Open;
  end; //with
end;
```

CloneCursor recebe três parâmetros. O Primeiro é o ClientDataSet origem, o segundo e o terceiro são chamados Reset e KeepSettings e são ambos booleanos. Quando Reset e KeepSettings são false as propriedades RemoteServer, ProviderName, IndexName, Filter, Filtered, MasterSource, MasterFields e ReadOnly são copiadas do ClientDataSet origem para o Clone. Quando Reset é True as propriedades do close são setadas para seus valores Default. Quando KeepSettings é True a propriedade values do close não é alterada.

### Salvando Dados Localmente

O componente ClientDataSet tem os métodos SaveToFile e LoadFromFile. SaveTofile salva o conteúdo da propriedade Data do ClientDataSet e qualquer índice que foi definido em tempo de projeto na propriedade IndexDefs para um arquivo local num formato proprietário. Se a extensão for XML somente os dados serão salvos no formato XML. LoadFromFile carrega os dados salvos de volta ao ClientDataSet.
A parte de ser muito fácil converter os dados para XML, estas funções tem muitos outros usos. Uma é criar aplicações de sincronismo que pode baixar um subset de registro do banco de dados no servidor, salvar os dados localmente, desconectar da rede, Incluir, apagar e alterar registros, reconectar a rede e gravar as alterações no banco de dados do servidor. Você pode também usar esta habilidade para reduzir o trafego em aplicações que rodam em redes lentas e que usa tabelas que são relativamente estáticas e contem moderado numero de registros. Apenas leia os dados das tabelas estáticas no ClientDataSet e salve eles localmente. A partir daí uses os dados das tabelas locais. Tudo o que você precisa é uma tabela de controle no banco de dados atualizada por uma trigger que mostre a data e hora que cada tabela foi alterada. Quando sua aplicação inicia, verifique a data e hora da ultima atualização de cada tabela e somente baixe do servidor se a tabela foi alterada desde a ultima vez que sua aplicação as carregou. Você pode também usar o ClientDataSet como um banco de dados desktop, sem utilizar banco de dados externos ou servidores.

### Usando ClientDataSets como tabelas em memória

Você pode usar o ClientdataSet sem conectar a um Provider. No design inclua os campos na propriedade FieldDefs usando o editor de propriedades. Então tudo o que você tem a fazer é clicar com o botão direito no ClientdataSet e selecionar Create Dataset no menu. Você agora tem uma tabela em memória com todas a características descritas acima. Você também pode incluir o FieldDefs em runtime usando o método FieldDefs.Add e chamar o método CreateDataSet do ClientDataSet para criar tabelas temporárias. Use estas tabelas temporárias para manipulação de dados, relatórios ou qualquer outra situação onde você deseja aumentar a performance e reduzir o trafego da rede armazenando os dados na memória.

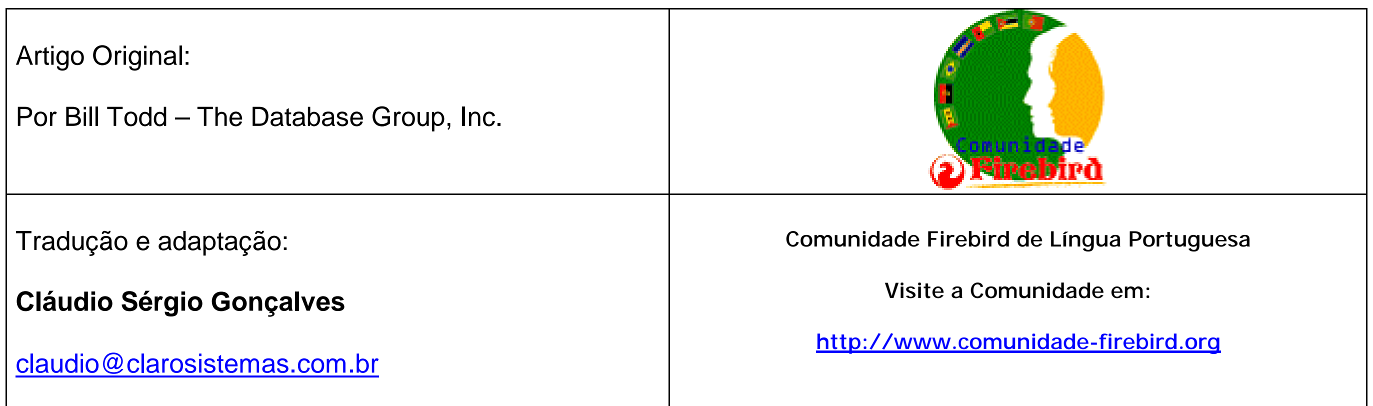
## Usando ClientDataSet

Se você usar ClientedataSets em sua aplicação você deve distribuir a MIDAS.DLL com sua aplicação. Esta DLL contem o código para suporte ao ClientDataSet. Você deve colocar a MIDAS.DLL no diretório da sua aplicação, no diretório Windows/System, no diretório Windows ou qualquer diretório que esteja no PATH.

# Sumário.

Usar ClientDataSets em suas aplicações Interbase oferece muitos benefícios. ClientDataSets pode reduzir tanto a carga do servidor quanto o trafego na rede. Ele também melhora a concorrência, pela diminuição do tempo que as transações ficam abertas. Somando a isso permite você ordenar e filtrar seus dados rápida e facilmente, permitindo fazer modelo de aplicações com sincronismo de dados e faz queries e Stored Procedures atualizáveis. Finalmente você pode usar ClientDataSets como tabelas temporárias e/ou tabelas na memória.

Copyright © 1994 - 2002 Borland Software Corporation. All rights reserved. Legal Notices Privacy Policy




| | |
|---|---|
| Artigo Original:<br><br>Por Bill Todd – The Database Group, Inc. | Comunidade Firebird |
| Tradução e adaptação:<br><br>**Cláudio Sérgio Gonçalves**<br><br>claudio@clarosistemas.com.br | Comunidade Firebird de Língua Portuguesa<br><br>Visite a Comunidade em:<br><br>http://www.comunidade-firebird.org |