Large Database applications in Delphi - Part 4: Extending TClientDataset

Michael Van Canneyt

March 16, 2014

Abstract

In this series of articles, some programming ideas will be discussed that can be applied when making large database applications in Delphi, i.e. database applications that have many screens, which operates on a database with possibly many tables.

1 Introduction

The `TClientDataset` component was introduced in early versions of Delphi primarily as a means of creating multi-tier database applications. In the latest versions of Delphi and Kylix, it is put forth as the primary method to present data to the user. In view of the many possibilities that the `TClientDataset` offers, this evolution can only be welcomed.

By itself, `TClientDataset` offers a rich set of features that makes its use attractive:

1. Support for multi-tier architecture, with the possibility to fetch data incrementally, on an as-needed basis.
2. Cached updates. Updates to the data can be cached and applied at will. A method to resolve conflicts in updates is present.
3. It can be used in a briefcase model: A `TClientDataset` component can store its data in a local file, and work with this file. Later, when an application server is present, updates can be applied on the server. This functionality can also be used to create a local, flat-file database.
4. Aggregate fields. The possibility exists to define fields based on aggregate expressions (sums, averages, counts etc) that are computed and maintained throughout the life-cycle of the dataset.
5. On-the-fly creation of indices. This means that the data in the client dataset can be sorted on almost any combination of fields.
6. Filtering using complex SQL expressions.
7. Master-detail relations between datasets based on link fields on the client.
8. Master-detail relations between datasets based on master-detail relations established on the server.
9. Internally calculated fields behave like calculated fields, except that their value is stored in memory. This means that the content of these fields can be used to filter on or be used in an index.

While the list of features should make it attractive for use as it is, there is some room for improvement; Mostly improvements which make it easier to program a `TClientDataset` component. In particular, some things can be done to reduce the amount of code that is still needed when using a `TClientDataset` component in a large database application.

In this article, some of these improvements will be discussed:

1. User-definable default values. `TClientDataset` uses the `DefaultExpression` property of `TField` to initialize field values when a new record is appended to the data; If the value of this property is changed after the dataset is open, the new value will not be used. A mechanism is implemented which observes changes to `DefaultExpression` and which saves these changes in a `TIniFile` object.
2. Master-Detail relations similar to the one that exist for `TQuery` objects. The Master-Detail relations offered by the standard TClientDataset have some drawbacks, which are circumvented in this implementation.
3. Automatically applied updates. By introducing a new boolean property 'AutoApplyUpdates', the behaviour of TClientDataset can be modified to updates to the data are sent at once to the server. It removes the need for a 'Apply updates' button or routine.
4. Automatic local caching of data. When a `TClientDataset` is used to lookup values from a dataset that changes little over time (e.g. a list of countries, or a list of towns), it makes sense to store the lookup data locally, and use the local copy of the data when looking up values. This is especially important in situations where the connection to the server is slow due to e.g. limited bandwidth.
5. Conversion of parameter values to field values. Often, in master-detail relationships (e.g. customers and orders), when a new record is created (a new order) , the value of the parameter used in the master-detail relation (customer number) is used as the value for a field (the customer-number field of the order). A mechanism is presented which automatically inserts the parameter value in a field when a new record is appended. This eliminates the need to insert these values manually in e.g. the `AfterInsert` event handler of the dataset.

More improvements can be made (and have been implemented by the author), but it would lead too far to discuss all possible enhancements. Also, the idea is not to demonstrate all possible ways to enhance `TClientDataset`, but to show that programming large database applications can be made easier and more efficient by implementing small application-specific enhancements to used components.

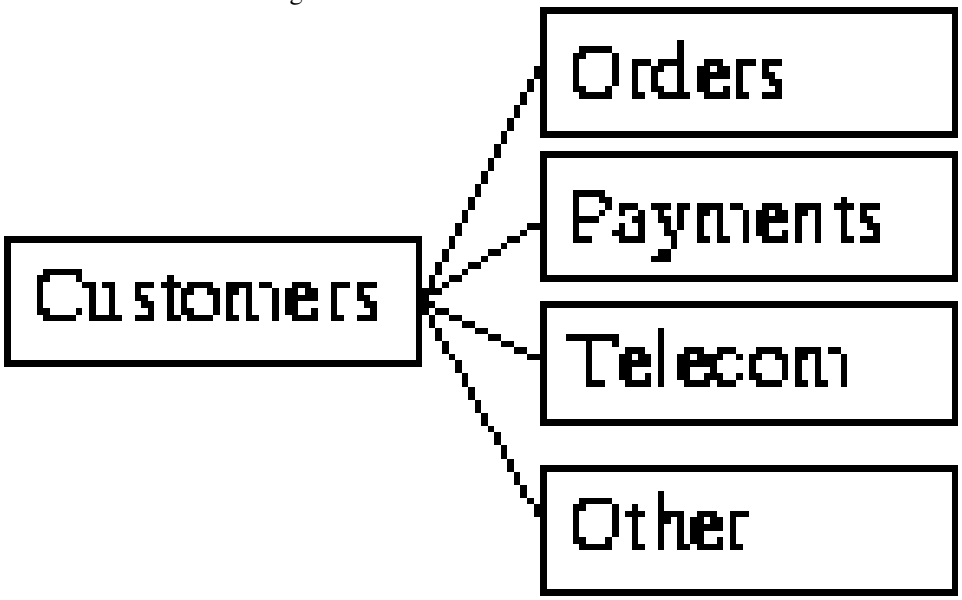
The sources for the presented enhancements is available on the CD that comes with this issue. A small demo Client/Server application is also present. It uses the standard BDE demo tables on the server to get data. A small README file explains how it should be used. The code as presented in the article is always filtered to show only one particular piece of functionality, for reasons of clarity. The code on the CD-ROM implemements all different functionalities combined.

2 A new TClientDataset Master-Detail mechanism

Suppose the database contains several tables, related to each other as e.g. in figure 1 on page 3. The relation is based on a unique customer identification (call it `CustNo`).

The data in this database must now be shown in several grids in such a way that scrolling in the list with customers will e.g. update the grid showing the orders from currently selected customer. The same for the list of payments, or the list of telecom data.

Figure 1: Tables in master-detail relation



The implementation of `TClientDataset` provides 2 ways to establish a master-detail relationship:

1. An implementation as is done in `TTable`: By defining link-fields, the detail dataset will only show records whose values are equal to the corresponding values in the master table.

   The disadvantage of this method is that ALL the records in the detail table are fetched from the server, and are then locally filtered so only the records corresponding to the current record in the master table are shown. When the detail table contains lots of records (e.g. customers orders), this is obviously very memory consuming, and will cause huge network traffic when the data is fetched from the server.

2. The second implementation is that the master-detail relationship is established on the server by traditional means (linked TTables or TQueries). The ClientDataset then fetches all the data from master and detail and stores the detail data in TDatasetField fields. A second client dataset can then be used to represent the detail data in the TDatasetField.

   This has the same drawback as the first method (unless incremental fetches are used to get the needed data), and, additionally, it means that all possible master-detail relations are hardcoded in the application server: if the application server doesn't have a provider which provides the needed master-detail relation, then the relation cannot be established on the client.

For the example database above, there are now 3 options available in Delphi to establish a master-detail relation:

1. Implement a provider which returns all the records in a table, and establish the required relation on the client using the linked-table mechanism.

2. Implement 3 providers, each returning the list of customers and the needed detail data combined.

3. Implement 1 provider, containing the list of customers and the 3 sets of detail data.

Each of these solutions suffers from the problems presented above. To circumvent these problems, a mechanism is implemented which works like the mechanism provided with the `TQuery` component:

A parametrized detail query can fetch values for the parameters from the fields in a coupled dataset (the master). When the current record in the master dataset changes, the parameter values for the detail query are updated and the query data is refetched using the new parameter values.

`TClientDataset` has support for parameters: It can send parameters to the dataset linked to its provider (which can be e.g a TQuery component). All that needs to be done is introduce a `DataSource` property, and make sure that when the current record changes in the dataset to which `Datasource` refers, the parameters for the detail dataset are re-sent to the provider, and the data corresponding to these parameters is fetched.

This means that for the database presented above, 4 providers must be set up: One which returns a list of customers, and which returns records from a detail table, related to 1 record in the master table (which record is specified by a parameter). This is similar to the linked-table approach, only the detail tables only return the needed records instead of all records from a detail table.

To illustrate this, imagine the following set-up. On the server, there are e.g. 3 data providers:

APCustomers Coupled to a `TQuery` component. It selects all customers:

```
select CustNo,Company,Addr1 from customer
```

APOrders Coupled to a `TQuery` component. Selects all orders from a customer. The customer whose orders are selected is determined by the `CustNo` parameter:

```
select * from orders.db where CustNo=:Custno
```

APPayment Also coupled to a `TQuery` component. Selects all payments from a particular customer.

```
select * from payments.db where CustNo=:Custno
```

On the server, no master-detail relation is established between these two latter queries and the query returning the list of customers.

On the client, there are 3 corresponding client datasets:

ADCustomers This dataset is coupled to the `APCustomers` provider on the server and will retrieve all customers.

ADOrders This dataset is coupled to the `APOrders` provider on the server. It has 1 parameter: `CustNo`.

ADPayments This dataset is coupled to the `APPayments` provider on the server. It has also 1 parameter: `CustNo`.

The clientdatasets are linked in a master-detail relationship on the `CustNo` parameter. Whenever the current record of `ADCustomers` changes, the `CustNo` parameter in the `ADOrders` or `ADPayment` dataset is updated with the new value and the data is refetched from the server.

The example above uses only a small number of possible master-details. In a large database application with many related tables (say 150 or more tables), the above presented mechanism offers more freedom in the choice of master-detail relations, and any number of detail

datasets can be coupled to a single 'master' dataset without loss of performance or without needing to code every possible master-detail relation on the server.

One can simply proceed as one would proceed when using ordinary `TQuery` components, i.e. supply a provider per needed detail. At run-time, the user can decide which detail he needs (e.g. orders, payments, telecom data) and only the needed data will be fetched from the server.

How can this be implemented ? Basically, the mechanism present in TQuery can be copied to a descendent of `TClientDataset`, let's call it `TAppDataset`. The values of the parameters must be fetched from a dataset associated with the `DataSource` property. This must be done at 2 points:

1. When the detail dataset opens.
2. When the current record in the master dataset changes. This can be done by closing and re-opening the detail dataset. The 'open' call will then reload the new values of the parameters.

What is needed is a `DataSource` property, and a `TDatalink` class descendent which will maintain the master-detail relation, and which will notify us when the master table has changed. This latter class is implemented in Delphi and is used to link components to a `TDataset` descendent.

The declaration of `TAppDataset` looks more or less as follows:

```
TAppDataset = Class(TClientDataset)
  FDataLink : TDataLink;
  procedure SetDatasource(const Value: TDataSource);
  function  GetTheDatasource: TDataSource;
  procedure RefreshParams;
  procedure SetParamsFromCursor;
  procedure SetParamsFromDataset(Dataset: TDataset);
Public
  Constructor Create(Aowner : TComponent); override;
  Destructor Destroy; Override;
  procedure OpenCursor(InfoQuery: Boolean);override;
Published
  property Datasource : TDataSource Read GetTheDatasource
                                    Write SetDatasource;
end;
```

When the dataset is opened, the parameters must be loaded from the dataset associated with the `DataSource` property. In `TClientDataset`, the parameters are sent to the provider in the `OpenCursor` method, which must therefore be overridden, so it loads the parameter values first before calling the provider:

```
procedure TAppDataSet.OpenCursor(InfoQuery : Boolean);

begin
  SetParamsFromCursor;
  inherited OpenCursor(InfoQuery);
end;
```

The `SetParamsFromCursor` method does the following:

```
procedure TAppDataset.SetParamsFromCursor;
```

```
begin
  if assigned(FDataLink.DataSource) then
    SetParamsFromDataset (FDataLink.DataSource.DataSet);
end;
```

And the `SetParamsFromDataset` does the actual work:

```
Procedure TAppDataset.SetParamsFromDataset(Dataset: TDataset);

var
  I: Integer;

begin
  if DataSet<>nil then
    for I := 0 to Params.Count - 1 do
      with Params[I] do
        If not Bound Then
          begin
          AssignField(Dataset.Fieldbyname(name));
          Bound := False;
          end;
end;
```

Note that parameters which are 'bound' will not be filled. (A parameter is bound when a value is assigned to it in code or in the object inspector in the IDE).

Changes in the master table are communicated to the detail dataset by the `FDatalink` field which contains an instance of `TAppDataLink`, a desdencent of the standard `TDetailDatalink` class as found in Delphi:

```
TAppDataLink = class(TDetailDataLink)
private
  FAppDataset : TAppDataset;
protected
  procedure ActiveChanged; override;
  procedure RecordChanged(Field: TField); override;
  function  GetDetailDataSet: TDataSet; override;
  procedure CheckBrowseMode; override;
public
  constructor Create(ADS: TAppDataset);
end;
```

The various methods in this class are called automatically when the data in the associated master dataset changes, e.g. when the user selects another record. These methods must be overridden so they can tell the detail dataset to refresh its parameter list and refetch the data from the server:

```
procedure TAppDataLink.ActiveChanged;
begin
  if FAppDataSet.Active then
    FAppDataSet.RefreshParams;
end;

procedure TAppDataLink.CheckBrowseMode;
begin
```

```
  if FAppDataSet.Active then
    FAppDataSet.CheckBrowseMode;
end;

constructor TAppDataLink.Create(ADS: TAppDataset);
begin
  inherited Create;
  FAppDataset:=ADS;
end;

function TAppDataLink.GetDetailDataSet: TDataSet;
begin
  Result:=FAppDataset;
end;

procedure TAppDataLink.RecordChanged(Field: TField);
begin
  if (Field = nil) and FAppDataSet.Active then
    FAppDataSet.RefreshParams;
end;
```

In the constructor of `TAppDataset`, an instance of the `TAppDatalink` class is created, and it is freed again in the destructor. The `RefreshParams` method, called by the `TDataLink` class, is responsible for the actual refreshing of the parameters. it does this by simply closing and re-opening the dataset:

```
procedure TAppDataSet.RefreshParams;

var
  DataSet: TDataSet;

begin
  if (FDataLink.DataSource<>nil) then
    begin
    DataSet:=FDataLink.DataSource.DataSet;
    if (DataSet<>nil) then
      if DataSet.Active and (DataSet.State<>dsSetKey) then
        begin
        Close;
        Open;
        end;
    end;
end;
```

This could be optimized by checking first if any of the parameter values has actually changed before reloading the data.

With this we have a working master-detail mechanism.

3 Converting parameters to field values

Now that the master-detail mechanism exists, it can be used. The details can be examined and edited. When editing a detail dataset, it is likely that a new record will be inserted.

For instance, a new order is appended to the list of orders for the customer. If the detail dataset has a parameter `CustNo` and a field `Custno` (needed to link it to the customer), when a new order is inserted for the customer, obviously the (required) `CustNo` field can be filled at once with the value of the `CustNo` parameter, resulting in a new order for the same customer.

Normally, this will be done in e.g. the `AfterInsert` event handler of the dataset, something like this:

```
Procedure TMyForm.ADOrdersAfterInsert(Sender : TDataset);

begin
 With ADOrders Do
  FieldByName('CustNo').asInteger:=
    Params.ParamByName('CustNo').AsInteger;
end;
```

If a lot of datasets in master-detail relation exist in the program, this can be automated, resulting in less code for the program.

The way to do this is to introduce a `ParamsToFields` property in the `TAppDataset` class. (It could be done for any `TDataset` descendent, this is not `TClientDataset` specific.) This is a `TStrings` property (it could be a collection as well) which contains a list of `Name=Value` pairs where `Name` is the name of a field, `Value` is the name of a parameter whose value must be inserted in the field. When a new record is inserted, this list is examined, and the field values are set up with the values of the parameters.

The relevant declaration in `TAppDataset` looks as follows:

```
TAppDataset = Class(TClientDataset)
Private
  FParamToFields : TStrings;
  procedure SetParamToFields(const Value: TStrings);
  procedure SetFieldsFromParams;
Protected
  Procedure DoAfterinsert; Override;
Published
  Property ParamToFields : TStrings Read FParamToFields Write SetParamToFields;
end;
```

The `FParamToFields` field is filled with a `TStringList` instance in the constructor of the class, and the property value is set in `SetParamToFields`. The real work is done in the `DoAfterInsert` method of `TAppDataset`. The `TDataset` implementation of this method is called after an insert occurs, and calls the `AfterInsert` handler, if there is one. The parameters will be inserted before the handler is called, so the programmer can override the parameter value if needed:

```
procedure TAppDataset.DoAfterinsert;
begin
  SetFieldsFromParams;
  inherited;
end;
```

The `SetFieldsFromParams` is actually nothing but a small loop through the list of `Name=Value` pairs in the stringlist:

```
procedure TAppDataset.SetFieldsFromParams;

Var
  I,L : Integer;
  S : String;
  P : TParam;
  F : TField;

begin
  For I:=0 to FParamToFields.Count-1 do
    begin
    S:=FParamToFields[i];
    L:=Pos('=',S);
    If L>0 then
      begin
      F:=Fields.FindField(Copy(S,1,L-1));
      System.Delete(S,1,L);
      P:=Params.FindParam(S);
      If (P<>Nil) and (F<>Nil) then
        F.Value:=P.Value;
      end;
    end;
end;
```

Note that in this routine, great care is taken that the list contains valid `Name=Value` pairs; This is a choice which was made. Another option would be to give an error message when some invalid value is encountered. (name of not existing field or parameter etc.)

To set this property, one could just use the IDE's built-in strings editor and enter the `Name=Value` pairs directly, or a specialized property editor can be written which presents lists of parameters and fields from which to compose the pairs. Such an editor is included in the sources for the design time package that is included on the CD-ROM. A screenshot of the property editor is visible in figure 2 on page 10

4 Default values for fields

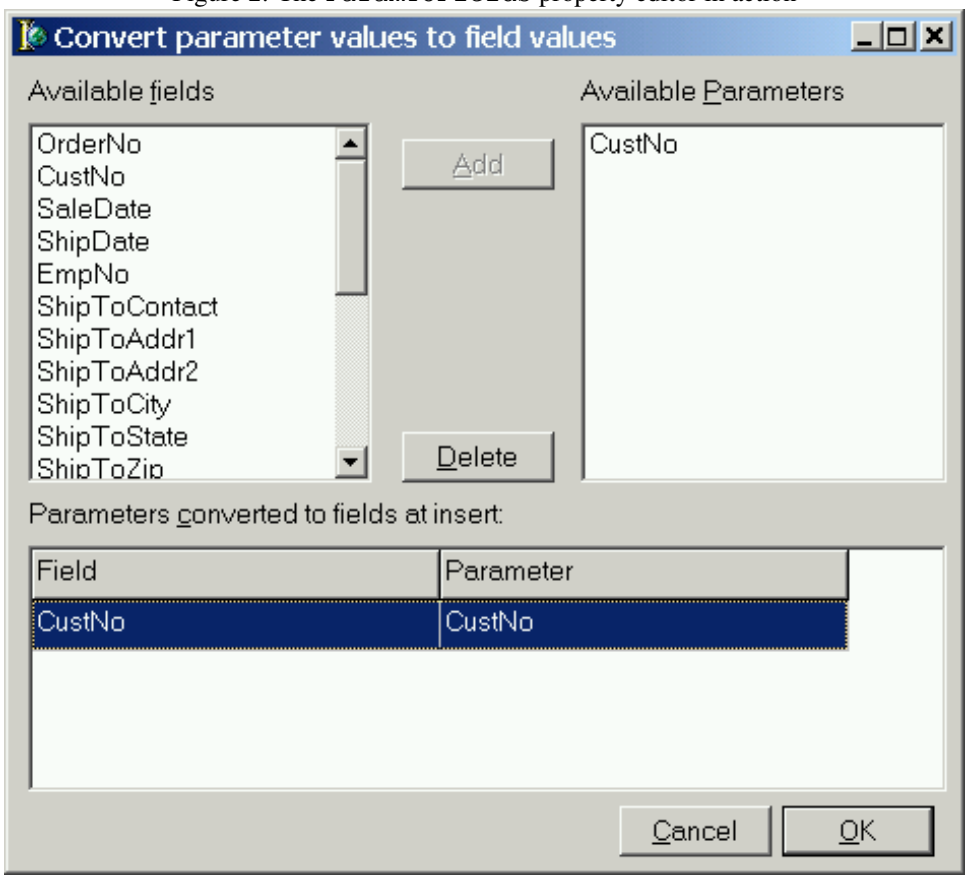
When a user must enter lots of similar data, the use of default values can be a great speed-up. Delphi's implementation of `TClientDataset` supports the `DefaultExpression` property of `TField`: The value of this expression is inserted into the field when a new record is created.

The implementation has a drawback, however: Changing the value of the `DefaultExpression` property after the dataset is opened has no effect. This means that the user cannot set the default values by himself. The values of the `DefaultExpression` properties as they were defined by the programmer are used - till the dataset is closed and re-opened, after which possible new values of `DefaultExpression` will be used.

Here an implementation is presented which allows the user to set (or clear) the default value of a field to its current value, and this default value will be effective at once. At the end of the session, the list of default values can be stored and retrieved at the next program startup.

To do this, an additional list is needed: The list of fields whose default value should be stored and set when a new record is inserted. Furthermore, a method is needed to store the current value of a field as the new default value, or to clear the default value. At insert, the default field values that were not yet applied by `TClientDataset` (i.e. the ones that

Figure 2: The `ParamToFields` property editor in action



were set after the dataset was opened) should explicitly be set. Finally, methods to store and retrieve the default settings are implemented.

The relevant declarations for all this are as follows:

```
TAppDataset = Class(TClientDataset)
Private
  FDefaultValues : TStringList;
  procedure SetDefaultValues(const Value: TStringList);
  procedure ApplyDefaults;
  procedure ResetDefaultFieldNames;
Private
  Procedure SetDefaultValue(FieldName : String);
  Procedure ClearDefaultValue(FieldName: String);
  Property DefaultValues : TStringList Read FDefaultValues
                                       Write SetDefaultValues;
end;
```

The `SetDefaultValue` sets the current value of a field as the default value. It does this by setting the `DefaultExpression` property to the current field value (as a string) and adding the name of the field to the `DefaultValues` list:

```
Const
  NumericalDataTypes = [ftSmallint, ftInteger, ftWord,
                        ftBoolean, ftFloat, ftCurrency,
                        ftBCD, ftLargeint];

  StringDataTypes = [ftString, ftDate, ftTime, ftDateTime,
                     ftFixedChar, ftWideString, ftTimeStamp];

procedure TAppDataset.SetDefaultValue(FieldName: String);

Var
  F : TField;

begin
  F:=Fields.FindField(FieldName);
  If F<>Nil then
    With F do
      begin
      If DataType in NumericalDataTypes then
        begin
        DefaultExpression:=AsString;
        FDefaultValues.Add('*'+FieldName);
        end
      else if DataType in StringDataTypes then
        begin
        DefaultExpression:=''''+AsString+'''';
        FDefaultValues.Add('*'+FieldName);
        end;
      end;
end;
```

The `NumericalDataTypes` and `StringDataTypes` constants determine which types of field values can be stored and how the defaultexpression is built. The fields are added to

the FDEfaultValues list with an asterisk (*) character before their name. The reason for this is that the default values which are set before the dataset is opened (their names are also in the FDefaultValues list) will be applied by TClientDataset itself, and there is no need to apply them again. When applying defaults when a new record is inserted, only the fields with an asterisk in front of them will be explicitly initialized.

Clearing a default value is simply the reverse operation:

```
procedure TAppDataset.CLearDefaultValue(FieldName: String);

Var
  F : TField;
  I : Integer;

begin
  F:=Fields.FindField(FieldName);
  If F<>Nil then
    F.DefaultExpression:='';
  With FDefaultValues do
    begin
    I:=IndexOf(FieldName);
    If (I=-1) then
      I:=IndexOf('*'+FieldName);
    If I<>-1 then
      Delete(i)
    end;
end;
```

When a new record is inserted, the default expressions must be applied to the record. It is again done in the DoAfterInsert method:

```
procedure TAppDataset.DoAfterinsert;
begin
  SetFieldsFromParams;
  ApplyDefaults;
  inherited;
end;
```

The ApplyDefaults method is nothing but a simple loop which sets the field value for all marked fields in the FDefaultValues list:

```
procedure TAppDataset.ApplyDefaults;

Var
  I : integer;
  Fn : String;

begin
  For I:=0 to FDefaultValues.Count-1 do
    begin
    FN:=FDefaultValues[I];
    If FN[1]='*' then
      begin
      System.Delete(FN,1,1);
      With FieldByName(FN) do
```

```
        begin
        If IsNull then
          If DataType in NumericalDataTypes then
            AsString:=DefaultExpression
          else if DataType in StringDataTypes then
            AsString:=Copy(DefaultExpression,2,
                           Length(DefaultExpression)-2);
        end;
      end;
    end;
end;
```

Note that the quote characters that must be present in the defaultexpression for string-like fields are cut off before setting the value.

If the dataset is closed and re-opened, the defaultexpressions properties will be applied by `TClientDataset`. In that case, they must not be applied anymore by the `ApplyDefaults` method. To avoid this, all asterisk markers are removed from the `DefaultValues` list when the dataset is opened. The proper place to do this is the `InternalOpen` method of `TClientDataset`:

```
procedure TAppDataset.InternalOpen;

begin
  inherited;
  ResetDefaultFieldNames;
end;

Procedure TAppDataset.ResetDefaultFieldNames;

Var
  S : String;
  I : Integer;

begin
  With FDefaultValues do
    For I:=0 to Count-1 do
      begin
      S:=Strings[i];
      If (Length(S)>0) and (S[1]='*') then
        begin
        System.Delete(S,1,1);
        Strings[i]:=S;
        end;
      end;
end;
```

Note that the above does not work with a sorted stringlist.

This is all that is needed to implement default values. In the example program accompagnying this article, the default value is set from the current field value by pressing F8 in the edit control for the field. It is cleared by pressing Shift-F8.

The implementation of this in a generic way is not presented here, the interested reader can consult the sources of the example program, the relevant event handler is the `OnKeyDown` method of the form.

The list of default field values can be saved (e.g. when a form is closed) and restored again (e.g. when a form is opened) with the `SaveSettings` and `RestoreSettings` methods:

```
procedure TAppDataset.SaveSettings(Ini: TCustomInifile);

Var
  I : Integer;
  S,Section : String;

begin
  With Ini do
    begin
    If Not ReadOnly then
      begin
      Section:=Name+'_Defaults';
      EraseSection(Section);
      for I:=0 to FDefaultValues.Count-1 do
        begin
        S:=FDefaultValues[i];
        If (S[1]='*') then
          system.Delete(S,1,1);
        WriteString(Section,S,FieldByName(S).DefaultExpression);
        end;
      end;
    Section:=Name+'_Preferences';
    If Assigned(FocusControl) then
      WriteString(Section,'FocusControl',FocusControl.Name)
    else
      DeleteKey(Section,'FocusControl');
    end;
end;
```

Note that the asterisk is stripped from the fieldname list before storing the list, and that default values with a CR/LF pair will give problems (This can be solved by using a slightly more sophisticated method). The `FocusControl` setting was discussed in an earlier article in this series, and will not be discussed again.

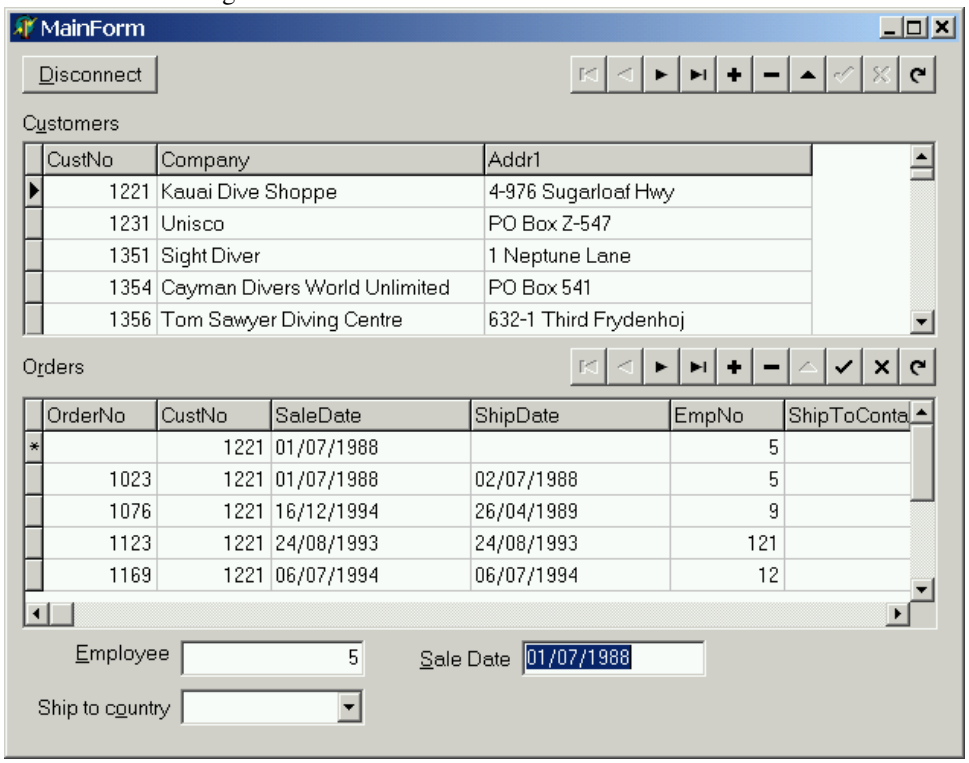
Restoring the settings is simply the reverse operation:

```
procedure TAppDataset.RestoreSettings(Ini: TCustomInifile);

Var
  I : Integer;
  S,Section : String;

begin
  With Ini do
    begin
    If Not ReadOnly then
      begin
      Section:=Name+SDefaults;
      ReadSection(Section,FDefaultValues);
      For I:=0 to FDefaultValues.Count-1 do
        begin
```

Figure 3: Default values and ParamToFields in action




```
        S:=ReadString(Section,FDefaultValues[i],'');
        With Fields.FieldByName(FDefaultValues[i]) do
          If DataType in NumericalDataTypes then
            DefaultExpression:=S
          else if DataType in StringDataTypes then
            DefaultExpression:=''''+S+'''';
        end;
      end;
    Section:=Name+SPreferences;
    S:=ReadString(Section,'FocusControl','');
    If (S<>'') then
      FocusControl:=Owner.FindComponent(S) as TWinControl;
    end;
end;
```

Note that these methods can only be used with persistent fields when the dataset is closed. If no persistent fields are used, the list of fields is not available when the dataset is not open; When saving and restoring these settings when a form is opened or closed, this is something which must be taken into account.

figure 3 on page 15 shows the example application in action after the 'append' button was clicked. The `EmpNo` field and `SaleDate` field have their default value filled in (obtained by presing F8 in the appropriate controls before the append operation). The `CustNo` field got its value from the `ParamToFields` property.

5 Automatically applying updates

When changes to a dataset of type `TClientDataset` are posted, the changes will not yet have ben applied to the provider from which the `TClientDataset` got its data. To apply the changes, a call to `ApplyUpdates` is needed. In a situation where the provider is present (e.g. when the client application is connected to the server application) the changes could be applied after each post. Applying changes to the database as soon as possible minimizes the risk of conflicts in cases where a lot of updates are done to the database.

To avoid having to call the `ApplyUpdates` method each time after changes have been posted, it is easier to override the post method and have it call `ApplyUpdates` automatically. To cater for situations where this behaviour is undesirable, a boolean property is introduced: `AutoApplyUpdates`; If its value is `True`, then `Post` will also call `ApplyUpdates`. If its value is `False`, then changes are not automatically applied to the server, e.g. in cases where one works with the briefcase model and the server is not present.

The implementation of the `AutoApplyUpdates` feature is quite easy. The relevant code is presented below:

```
TAppDataset = Class(TClientDataset)
Private
  FAutoApplyUpdates : Boolean;
Public
  Procedure Post; override;
Protected
  Property DoAutoApplyUpdates; virtual;
Published
  Property AutoApplyUpdates : Boolean Read FAutoApplyUpdates
                                      Write FAutoApplyUpdates;
end;

procedure TAppDataset.Post;
begin
  inherited;
  If AutoApplyUpdates then
    DoAutoApplyUpdates;
end;

procedure TAppDataset.DoAutoApplyUpdates;

begin
    ApplyUpdates(0);
end;
```

Introducing a second property, the `DoAutoApplyUpdates` procedure could be enhanced so it applies updates every `N` updates:

```
procedure TAppDataset.DoAutoApplyUpdates;

begin
  If (ChangeCount>=N) then
    ApplyUpdates(0);
end;
```

The standard `ChangeCount` property indicates how many unapplied changes are still waiting to be applied on the server. Note that in this case, it is necessary to check whether

there are still unapplied changes left when the dataset closes or the connection to the server is about to be broken.

6 Transparant caching

In relational databases, often one needs to use a lookup dataset to be able to look up values for some field. For instance a town, a country, a customer number, in general any list of parameters that is saved in the database. When coding an application that uses lots of lookup datasets, it makes sense to cache this data on the client if the communications line is slow (e.g. a telephone connection). Needless to say that the caching is only useful if it is known that the lookup data doesn't change too often, as e.g. in a list of countries or towns. Also if the same lookup data is used in more than one form, the caching of this data has advantages.

A `TClientDataset` can save its data locally to disk. This feature can be used to implement caching. The mechanism is quite easy: When the dataset is opened, it checks whether a cache file is present. If a file is present, it is used to load the data. If a cache file is not present, the data is fetched from the server, and saved to disk at once. The filename for the cache file is determined automatically from the providername property and the values of parameters (if any).

Obviously there must be methods to clear the cache or refresh the cache. If changes have been applied to the lookup list, then the local cache must be updated to refresh these changes.

The caching is implemented using the following properties and methods:

Cache This boolean property controls the activation of the caching mechanism. Caching will only be performed if its value is `True`

GetCacheFileName Returns the name of the cache file.

RefreshCache This is done by clearing the cache and closing and re-opening the dataset.

ClearCache This simply deletes the cache file.

The relevant declarations for the caching is shown below:

```
TAppDataset = Class(TClientDataset)
Private
  FCached : Boolean;
  function  GetCacheFileName: string;
Protected
  procedure InternalClose; override;
  Procedure InternalInsert; override;
  Procedure DoAfterinsert; Override;
  procedure InternalDelete; override;
  procedure InternalRefresh; override;
  Procedure InternalOpen; override;
Public
  procedure ClearCache;
  procedure RefreshCache;
  procedure OpenCursor(InfoQuery: Boolean);override;
Published
  Property Cached : Boolean read FCached write FCached;
end;
```

In the `OpenCursor` method (called when the dataset is opened) we set the `FileName` property if need be. If the file exists, it will be used to load the data. When the data is loaded (be it from the file or from the provider) the existence of the cache file is checked. If it does not exist, the cache file is created by calling the `SaveToFile` method:

```
procedure TAppDataSet.OpenCursor(InfoQuery : Boolean);

begin
  SetParamsFromCursor;
  If Cached then
    FileName:=GetCacheFileName
  else
    FileName:='';
  inherited OpenCursor(InfoQuery);
  if (FileName<>'') and not FileExists(FileName) then
    SaveToFile (FileName);
end;
```

When refreshing the dataset, it is necessary to clear the cache. To do this, the `InternalRefresh` method is overridden so it clears the cache.

```
procedure TAppDataset.InternalRefresh;

begin
  if Cached then
    RefreshCache
  else
    Inherited InternalRefresh
end;
```

The `RefreshCache` method closes the dataset, clears the cache and re-opens the dataset.

```
procedure TAppDataset.RefreshCache;

var OldCaching : boolean;
    opened : boolean;

begin
  OldCaching := Cached;
  Opened := Active;
  FCached := False;
  if Opened then
    close;
  ClearCache;
  Data := null;
  Cached := OldCaching;
  if Opened then
    open;
end;
```

The `ClearCache` method does nothing except deleting the cache file, if it exists:

```
procedure TAppDataset.ClearCache;
```

```
Var
  FN : String;

begin
  FN:=GetCacheFileName;
  If (FN<>'') and FileExists(Fn) then
    If not DeleteFile(FN) then
      Raise Exception.CreateFmt('Could not delete cache file %s',[FN]);
end;
```

When closing the dataset, the cache file must be written to disk, in case the `Cached` property was set after the dataset was opened:

```
procedure TAppDataset.InternalClose;
begin
  If Cached then
    FileName:=GetCacheFileName;
  inherited;
end;
```

Finally, the `GetFileName` call returns a filename for the clientdataset. It does this, based on the name of the provider the clientdataset is connected to, and the values of any parameters that are passed to the providers:

```
function TAppDataset.GetCacheFileName : string;

  function ParamToString(param : TParam) : string;
  begin
    Result:=Trim(Param.AsString);
    Result:=StringReplace(Result,'\','-',[rfReplaceAll]);
    Result:=StringReplace(Result,'/','-',[rfReplaceAll]);
  end;

var
  I : integer;
  s : string;

begin
  Result:=SCacheDir;
  If (Result<>'') then
    begin
    S := ProviderName;
    for I := 0 to Params.count-1 do
      s := s + '-' + paramToString(Params[I]);
    Result := Result + S + SCacheExtension;
    end
end;
```

The rationale behind this is that several cached clientdatasets (on various forms) that use the same provider will use the same cache file. The parameters are needed because different parameters will result in different data being returned: The data should be uniquely determined by the combination of the providername, and parameter values. If clientdatasets that use the same provider, but with different parameters would use the same cache file, the result would be that the first opened dataset would fill the cache file with data correspond-

ing to its parameter values, and the second clientdataset will use wrong values if it has a different value for the parameter.

For example, suppose we have a table with a list of towns; each town has a reference to a country using the ISOCode of the country. A provider provides a list of towns in a certain country (e.g. using a parameter 'Code'). If two datasets get their data from this provider, one with parameter B (belgium), the other using parameter D (Deutschland), were to use the same cache file, then the first opened dataset (e.g. the one with German towns) would fill the cache file, and the second (the one expecting towns in Belgium) would use the cache file with the data from the first dataset (i.e. the German towns). By integrating the value of the parameter, the problem is solved because the two datasets will use different cache files.

The example program contains a lookup control which presents a list of countries. The list of countries is obtained from a cached dataset. The first time the program is run, a file will be created in a subdirectory 'Cache' and filled with the data from the application server. The second time the program is run, the data is not fetched from the server, but is read from the cache file.

To check this, after the program has run the first time, one can open the cache file using any editor (it is a text file, in XML format) and remove any entry from the file. When the program is run a second time, the deleted entry will not be present in the list, showing that the data was read from the file and not fetched from the server.

The presented mechanism can be enhanced by introducing some audit fields (time of last modification) in the database. By comparing the modification time of the cache file with the audit fields, one can decide whether it is necessary to refetch the cached data from the server.

7 Conclusion

Some improvements to TClientDataset were presented. Many others can be implemented: Support for duplication of a record, recalling of last entered values in a control. Support for automatic fetching of values for autoincremental fields; support displaying the dataset status in a status bar etc. The idea is that in a large database application, development time can be reduced drastically by implementing a series of small enhancements, which can be used throughout the whole application, giving the application a uniform look and feel without the need for additional effort from the programmer.