Under Construction: Nested Tables

by Bob Swart

Like any piece of technology related to tables in Delphi, the title of this month's column could also be *Nested DataSets*: it's actually more accurate, as we'll see later, though the relevant VCL component is called `TNestedTable`.

TNestedTable

The `TNestedTable` component is one on the `Data Access` tab of the Delphi 4 Client/Server component palette (it's not on the Delphi 4 Professional palette, although some claim they can create it dynamically if the `DBTables` unit is included in the `uses` clause).

According to the official definition (from the help), *'a nested table is a dataset component that encapsulates a database table that is nested as a field within another table'*. After reading this definition several times we're left with a question: how do we *make* a nested table? Or, how does one get a database table that is nested as a field within another table?

The help adds the following. First: *'Use* `TNestedTable` *to access data contained in a nested dataset.'* (that's helpful... Not). Second: *'A nested table inherits BDE functionality from* `TBDEDataSet` *and so uses the Borland Database Engine (BDE) to access the nested table data.'* This is not even correct, as it turns out we only need the BDE when working with the `TNestedTable` component, not when working with a 'nested table' itself, something which will become clear in a moment. And finally: *'A nested table provides much of the functionality of a table component, except that the data it accesses is stored in a nested table.'* Now why do I suddenly feel locked into a moebius ring?

Nested Tables

So, before we can use a `TNestedTable` component, it appears we must first fabricate a 'nested table' itself. This should then be connected to the `DataSetField` property of the `TNestedTable` component. In fact, the `TNestedTable` component shows all available `DataSet` fields and `Reference` fields in a drop-down combobox for this `DataSetField` property. Of course, if we just drop a `TNestedTable` on an empty form or data module, we won't see anything in this list. How to get items (`DataSetFields`, or nested `DataSets`) inside this list, that's what we're looking for now.

Oracle 8

Well, don't look too long, as it appears that nested tables are a specific feature of Oracle 8. Nested datasets represent the records of an Oracle 8 nested detail set.

Unfortunately, Delphi 4 doesn't offer us the ability to actually create Oracle 8 tables with these nested dataset fields, although we can display and modify data from existing Oracle 8 dataset fields using nested datasets. In other words: as far as I can see now, the `TNestedTable` component can only be used for Oracle 8 tables that contain a nested detail set, and for nothing else (if someone else knows a way to use the `TNestedTable` component without Oracle 8, let me know!).

Master-Detail

The conclusion we arrived at above could mean the end of this month's column. Fortunately, it only means the end for the `TNestedTable` component for this month, and certainly not the end of the concept of nested tables itself. Apart from Oracle 8 nested detail sets, the idea of nested tables is also used by the developers at Inprise to offer a more powerful way of dealing with master-detail relationships in a multi-tier distributed application.

The traditional (standalone and client/server) way of defining master-detail relationships is based on the detail table connecting to a datasource (pointed to the master table), defining a relation between one or more fields from the detail table and the master table. For example, drop two tables on a form, set the `DatabaseName` to `DBDEMOS`, and the `TableName` properties to customer.db and orders.db respectively. Now, drop a `DataSource` on the form, set its `DataSet` property to the customer.db table component, and set the `MasterSource` property of the orders.db table component to this `DataSource` (so customer.db is the 'master' of the orders.db table). Now, click on the ellipsis next to the `MasterFields` property of the orders.db table. We'll get the Field Link Designer dialog, in which we can specify the index (for the orders.db table) to use, and the fields to connect the orders.db detail table to the customer.db master table. In our case, we must use the `CustNo` index, to link the `CustNo` fields from both tables to each other.

For a standalone or client/server application, this works fine. However, for a multi-tier application (or more general, in an N-Tier environment where N is bigger than 2), where the data for the tables is provided by a database server, this scenario has at least two serious drawbacks.

First of all, the detail table must fetch and store all its records from the database server, even if only a few of the detail records are actually needed at the client side (after the master-detail relationship has been established). So, a potentially large number of records are sent over for nothing. Whilst in the current era of computing CPU cycles may no longer count, bandwidth cycles are more important than ever. Wasting bandwidth by sending over the entire detail table before determining the detail records we actually need is a serious crime in this internet age (believe me, it will take at least

another decade before bandwidth is as free as CPU speed and memory are now). Of course, this problem can be overcome by using parameters, sent from the client to the server, but this involves more work and can introduce bugs that are hard to trace.

The second drawback of the traditional method of defining master-detail relationships has to do with the fact that it's more difficult to apply updates using client datasets. This is caused by the fact that the `ClientData` component doesn't apply updates for multiple tables in a single transaction, but on a dataset by dataset level (ie we must make a separate call to `ApplyUpdates` for the master and detail tables, in which the order may be important).

In N-tier applications (where N is bigger than 2), we can now avoid these problems by using nested tables to represent the master-detail relationship. All we need to do is to define a master-detail relationship between the tables on the remote data server (not on the client), drop a `Provider` component and export the master table only (not the detail table). The trick is that the master table will automatically include a `DataSetField` for the `Detail` records. And indeed only for those detail records that are relevant for the current master record.

When clients call the `GetRecords` method of the provider, it automatically includes the detail datasets as a `DataSet` field in the records of the data packet. When clients call the `ApplyUpdates` method of the provider, it automatically handles applying updates in the proper order.

Remote Master-Detail

So, we just need to repeat what we did for the client side a moment ago, but this time on a `RemoteDataModule`. Drop two tables, a datasource and a provider component, create the master-detail relationship (for the customer.db and orders.db tables), connect the provider component to the customer.db table, and export it from the remote data module.

Note that full source code is on the disk, as usual, but the important unit for the Remote Data Module should contain the declarations and code shown in Listing 1 when you're finished.

TClientDataSet

Now that we've finished the server side, let's focus on the client side, where we use a `TClientDataSet` to connect to our `CustomerOrders` data. Start a new application, drop a `DCOMConnection` component, and set the `ServerName` to the name of the remote master-detail app (note that we could also have created a CORBA Data Module in combination with a `CorbaConnection` component, but it's just a bit easier to show the DCOM example). Next, drop one `ClientDataSet` component, connect it to the `DCOM Connection` component (using its `RemoteServer` property), and select the (only) provider for the nested `CustomerOrders` table.

We now have a thin client (ie it doesn't need the BDE), with a `ClientDataSet` component that contains a (remote) nested table. In order to be able to actually use the nested dataset (ie the detail records), we must create a persistent `DataSet` field for the nested data. This sounds more difficult than it is: just double click on the `ClientDataSet` to start the Fields Editor (at design-time), right click in the Fields Editor and select `Add All Fields`. This will create persistent fields for every field, including a `DataSetField` for the nested detail table.

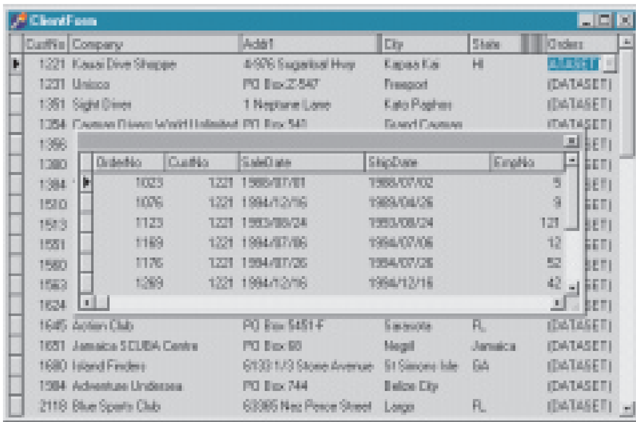
We still cannot use this `DataSetField` right away, however. There are a number of ways to 'drill-down' into the nested table. One way, often demonstrated at seminars by people like Charlie Calvert, is to use a `DBGrid` control connected by a `DataSource` to the `ClientDataSet`. The `DBGrid` component has the ability to recognise a `DataSetField` and displays it in a special way including an ellipsis (...) when we select that particular cell of the `DBGrid` at runtime. If we click on the ellipsis, a pop-up window appears that holds another `DBGrid` control with the detail records (ie the entire contents of the nested dataset we just selected).

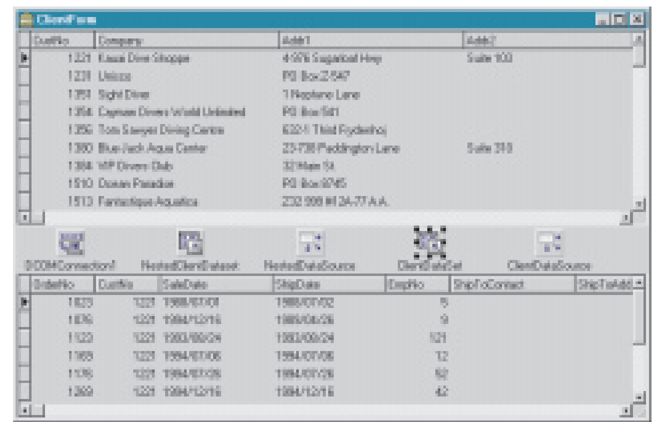
Note that we can also do this programmatically by calling the `DBGrid.ShowPopupEditor` method, with as argument the `DataSetField` that we want to show in the pop-up Window: see Figure 1.

I must admit, this is a spectacular effect when you see it for the first time. Maybe it's fun the second time as well, but it gets boring the third time. And let's be honest: would you want to tell your client that he has to click on the ellipsis to get the detail data? Just because it's so much easier for you to perform updates (we only need to call `ApplyUpdates` on the master `ClientDataSet`!). We want to offer our client a solution where both the master and detail can be viewed at the same time. Charlie's workaround is to put a panel on the lower half of the form which holds the master `DBGrid`,

Listing 1

```
unit ServerU;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComServ, ComObj, VCLCom, StdVcl, BdeProv,
  DataBkr, DBClient, NestedSrv_TLB, Provider, Db, DBTables;
type
  TNestedDataModule =
    class(TRemoteDataModule, INestedDataModule)
    Customer: TTable;
    Orders: TTable;
    dsCustomer: TDataSource;
    CustomerOrders: TProvider;
  private
  public
  protected
    function Get_CustomerOrders: IProvider; safecall;
  end;
var
  NestedDataModule: TNestedDataModule;
implementation
{$R *.DFM}
function TNestedDataModule.Get_CustomerOrders: IProvider;
begin
  Result := CustomerOrders.Provider;
end;
initialization
  TComponentFactory.Create(ComServer, TNestedDataModule,
    Class_NestedDataModule, ciSingleInstance, tmApartment);
end.
```





➤ *Above left: Figure 1*
Above right: Figure 2

making sure the panel can accept a dockable control (like the detail `DBGrid`). This is also cheating a bit and I still wouldn't want to tell my client to click on the ellipsis first, then drag the pop-up window to the empty panel under the first `DBGrid`, and so on.

No, there has to be a simpler way. And as usual with Delphi, there is. For that, we need to drop yet another `ClientDataSet` on the form, and this time connect the `DataSetField` property to the (only) persistent `Orders` `DataSetField` which is called `NestedClientDatasetOrders` in my example.

Note that this `ClientDataSet` is not directly connected to a remote server, but indirectly, since it gets its data from the nested dataset that the first `ClientDataSet` received from the remote data server. Both `ClientDataSet` components should now be connected to a `DataSource` and a `DBGrid` to show the master and detail in a single thin client form: see Figure 2.

I think this is a good solution for displaying and updating master-detail relationships. Sometimes displaying the detail in a pop-up window may be what you need, sometimes my solution with a secondary `ClientDataSet` is better.

The problem with updating the master-detail relationship is solved by the fact that we now have only one call to `ApplyUpdates` to make (from the `ClientDataSet` that is directly connected to the remote server), which automatically updates the entire nested table.

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

➤ *Listing 2*

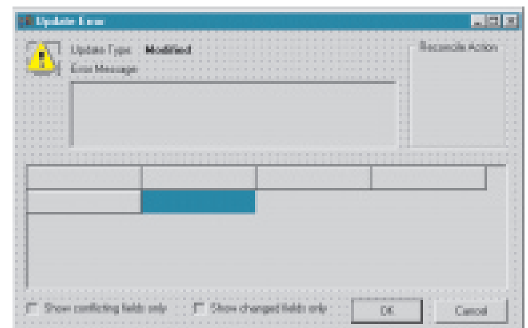
Error Handling

As a final topic this month, let's take a look at the result of calling `ApplyUpdates`. This method takes one argument: the number of errors we 'tolerate' before generating a reconcile error event. Usually, I would pass the value -1, to indicate that I don't want to tolerate any update error whatsoever. However, all kinds of update errors can occur, for example if someone else made a change to one or more records in the master-detail relation on the server. In those cases, we get an `OnReconcileError` event at the `ClientDataSet`, which we can handle by using a prepared form from the Object Repository. Do a `File | New`, go to the `Dialogs` tab, and select the `Reconcile Error Dialog`, Figure 3.

The resulting dialog also contains the logic to deal with the errors themselves, including `Actions` such as `Skip`, `Abort`, `Merge`, `Correct`, `Cancel` and `Refresh`.

To use the dialog, we must include the newly generated unit, and write the single line of code shown in Listing 2 for the `OnReconcileError` event handler of the `ClientDataSet` component.

That's it! A powerful and handy chunk of technology available for our use with nested tables.



➤ *Figure 3*

Next Time

We've discovered nested tables, seen how to use them for master-detail relations in multi-tier applications, and examined both displaying and update techniques.

A final warning when using nested tables on the client side: `TDBGrid` may have difficulty displaying data when the associated dataset consists of tables nested too deeply: generally, 5 or more levels of nesting, though I wouldn't even dream of using 5 levels...

After all this nesting, it's time to unravel things again. And what better way to do this than experimenting with some Delphi debugging techniques, including remote debugging. *So stay tuned!*

Bob Swart (aka Dr.Bob, visit www.drbob42.com, email him at drbob@chello.nl) is a technical consultant and webmaster using Delphi, JBuilder and C++Builder, and a freelance technical author.