Persistent Fields: Friend Or Foe?

by Guy Smith-Ferrier

Persistent fields have been a feature of Delphi since its inception. In this time there has been little change to their operation, yet their use, or lack of use, is often fiercely debated.

This article is not an introduction to persistent fields, but an analysis of their strengths and weaknesses, dispelling a myth or two, and applying some fresh information in Delphi 5.

Basic Use

All datasets have an array called `Fields` which usually contains a `TField` object for every column in the result set. `Fields` can be accessed either by their ordinal position:

```
Table1.Fields[2].AsString :=
  'Sirius Cybernetics';
```

or using the `FieldByName` method:

```
Table1.FieldByName(
  'Company').AsString :=
  'Sirius Cybernetics';
```

or using the `TDataSet`'s default variant array:

```
Table1['Company'] :=
  'Sirius Cybernetics';
```

A persistent field is a field added by the Fields Editor at design-time for the purposes of simplifying and speeding the development process. Double click on any dataset object (BDE, ADO, InterBase Express, `TClientDataSet`) and you will see the Fields Editor. Right click the Fields Editor, select `Add all fields` and persistent fields will be added to the Fields Editor. If you take a look at the underlying form's source code you will see a new field has been added to the form for each field added to the Fields Editor:

```
TForm1 = class(TForm)
  Table1: TTable;
  Table1CUST_ID:
    TSmallintField;
  Table1COMPANY: TStringField;
  Table1PHONE_NO: TStringField;
private
...
```

That's all there is to it. You are now ready to reap the rewards and pay the price of persistent fields.

The Rewards

The simplest and greatest benefit of persistent fields is compile-time syntax checking. To use a persistent field you refer to the field added to the form's class:

```
Table1COMPANY.AsString :=
  'Sirius Cybernetics';
```

This scores above all of the previous methods of accessing a field because it gives you compile-time syntax checking on the field name. All of the other methods check the field name at runtime. As a general rule, if you are faced with two solutions where one gives compile-time syntax checking and the other gives runtime syntax checking, and all other factors are equal, then choose the solution with compile-time syntax checking.

The second benefit is the ability to use the Object Inspector to set the field's properties and events. All Delphi programmers know how easy it is to click in the Object Inspector, or double click, to enter an event and it is this ease of use which is one of Delphi's strengths. However, it is not true to say that if you don't use persistent fields you don't have access to a field's properties and cannot set a field's events. All persistent fields inherit from `TField`, which contains most of the properties, methods and events used in its descendants. In particular, you can still set events for fields, but without the aid of the Object Inspector you are forced to set this in code (see Listing 1).

Although this is certainly possible, given a choice between the two techniques, setting properties and events using the Object Inspector is certainly easier.

The third benefit is the ability to drag persistent fields from the Fields Editor and drop them on the form. This truly is Rapid Application Development, as all the fields are neatly laid out on the form. There's more on this subject later. Incidentally, if you like this feature but decide not to use persistent fields, you can still use it by simply adding persistent fields to the Fields Editor, dragging and dropping them on to the form, and then deleting the persistent fields.

The fourth benefit is the option to take advantage of Delphi's Data Dictionary. I won't cover the Data Dictionary here because Steve Troxell expertly explained it in Issue 41. However, suffice to say that the Data Dictionary is intended to provide developers with the ability to centralise the setup of all of the fields in one or more applications. However, as the Data Dictionary is based on aliases, it can only be used with the BDE datasets, not with `TClientDataSet`, ADO Express datasets or InterBase Express datasets.

Lookup Fields

Another benefit of persistent fields is that their presence allows you to create lookup, calculated and aggregate (`TClientDataSet` only) fields. Lookup, calculated and aggregate fields are all created by right clicking the Fields Editor, selecting `New field` and choosing the relevant field type. Calculated fields require you to write code to calculate the new field's value in the dataset's `OnCalcFields` event (the same event is shared between

➤ *Listing 1*

```
procedure TForm1.FormCreate(
  Sender: TObject);
begin
  Table1.Fields[0].OnValidate :=
    ValidateCustNo;
end;
```

all calculated fields). Aggregate fields allow you to sum, count, average, min or max a group of fields. This is an incredibly useful feature but it is only available in TClientDataSet. It is especially useful for totalling columns in a grid or keeping batch totals accurate in batch mode data entry.

A lookup field is a lookup from a field of one table into another table. For example, using data from the IBLOCAL alias, an EMPLOYEE table might have a lookup field called DEPARTMENTNAME which is a lookup from the DEPT_NO field in EMPLOYEE into the DEPARTMENT table. The corresponding field in the DEPARTMENT table is DEPT_NO and the returned field is DEPARTMENT. The fact that the field is called DEPT_NO in both the EMPLOYEE table and the DEPARTMENT table is simply a result of the naming convention used in this database: it isn't a requirement of Delphi's lookup fields. The new lookup field acts like any other field and, unlike a calculated field, is a read/write field.

As you can see, lookup fields serve a similar purpose to SQL JOINs and therefore a comparison is in order. Let's start with an equivalent SQL equi-join:

```
SELECT EMPLOYEE.*,
  DEPARTMENT.DEPARTMENT FROM
  EMPLOYEE, DEPARTMENT WHERE
  EMPLOYEE.DEPT_NO=
  DEPARTMENT.DEPT_NO
```

This SQL gets executed on the server (assuming a client/server DBMS) and the resulting data is sent back across the network to the client (ie the BDE, ADO or InterBase Express). To help us compare the two techniques, let's do an example comparing the amount of data sent across the network. For simplicity, let's say that each EMPLOYEE record is 100 bytes and the department name is 25 bytes. So 100 records total 25,000 bytes.

Lookup tables clearly don't work the same way. However, the problem with the popular misconceptions about how lookup fields really work have their roots in the same misconceptions about most database features in Delphi. The misconceptions are caused by the fact that TTable and TQuery solve their problems in a fundamentally different way. In general, TQuery attempts to cache its data so that once a record has been read any further attempt to read it is read from its own internal cache. TTable, on the other hand, generally does not assume state. As a result it frequently reads and re-reads data. But this is not the place to restart 'The Great TTable Versus TQuery Debate'. Let's start with TQuery.

A lookup field which gets its data from a TQuery works very well. The TQuery is opened when its dependent table is opened. All its records are read into its internal cache. Thereafter, all lookups into the TQuery are serviced from its cache and no further access to the DEPARTMENT table is made. So, continuing the example, the amount of data sent across the network would be 100 EMPLOYEE records (100 records * 100 bytes) plus all the DEPARTMENT records. As the TQuery can specify exactly what fields are required (eg SELECT DEPT_NO, DEPARTMENT FROM DEPARTMENT) then this would be 21 records * 29 bytes. Thus the total data sent across the network would be 100,609 bytes (ie 20% less data than a JOIN). Of course, some programmers might use SELECT * FROM DEPARTMENT, in which case the total would be 101,596 bytes, which doesn't change the conclusion.

TTable, however, works quite differently. When the dependent table is first opened the TTable is also opened. Only the first record is retrieved by the application (although the DBMS has generated the entire result set anyway). Thereafter, each time the dependent table's record pointer is moved, the TTable has to execute a new SELECT statement to retrieve the lookup field's value:

```
SELECT DEPT_NO, DEPARTMENT,
  HEAD_DEPT, MNGR_NO, BUDGET,
  LOCATION, PHONE_NO  FROM
  DEPARTMENT WHERE DEPT_NO=?
```

Since TTable doesn't cache this information, if you revisit a record TTable has to execute the same SELECT statement it executed the first time you visited the record. It's difficult to say how much information passes across the network, because it depends on how much activity occurs in the dependent table. Given enough time, however, it would eventually exceed the SQL JOIN.

So what do we learn from this? Firstly that TQuery provides much more efficient lookups than TTable. Secondly, Delphi's lookup fields have the potential to give better performance than a regular JOIN, because less information passes across the network. However, there is a far more compelling reason to use Delphi's lookup fields instead of a JOIN: to make a JOIN read/write (using the BDE) you have to use either cached updates or TClientDataSet. If you use a lookup field instead of a JOIN the result set stands a much better chance of being read/write and therefore maintenance is simpler.

Just before we leave lookup fields, let's return to the drag and drop I talked about earlier. If you drag and drop a lookup field onto your form, the standard Fields Editor drops a TDBLookupComboBox on your form which is already configured correctly.

TTable And SELECT

OK, now we know all about persistent fields, let's dispel a few myths.

One of the criticisms of TTable in 'The Great TTable Versus TQuery Debate' is that because TQuery allows the programmer to specify the fields to be returned in the result set the programmer can exclude unwanted fields and therefore less data will be passed across the network. The volume of data in the result set is one of the biggest factors affecting the performance of a SELECT statement. As you cannot define the SELECT statement yourself in TTable, it is less efficient. Programmers often defend TTable by saying that although you don't have access to the SELECT statement itself you can get TTable to retrieve just the fields you want by using persistent fields. If only this were true. Sadly, it is not, and this puts rather a large nail in the coffin of TTable. Incidentally, it is easy enough to determine whether this is true or not using SQL Monitor to show the SELECT statement which is generated (SQL Monitor is Delphi's forgotten tool and all BDE developers should try it out).

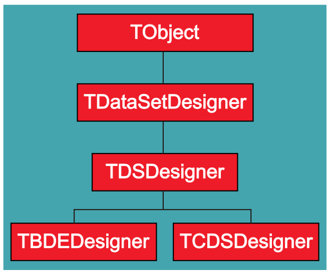
In defence of these programmers, though, it is easy to adopt this (false) belief. Here is an extract from Delphi's help on the benefits of persistent fields: 'You can restrict the fields in your dataset to a subset of the columns available in the underlying database.' The statement is true but misleading. It is referring to a logical restriction on which fields are visible and can be accessed; it does not, however, refer to the fields in the SELECT statement.

Performance

One of the benefits often touted for persistent fields is that they are faster than other methods of accessing fields. This is simple enough to test (see Listing 2).

First the test is run using the persistent field, and then using FieldByName. Although this is not the most scientific of tests, the conclusion is still accurate. Using the persistent field the test takes 6 seconds and using FieldByName it takes 9 seconds. So, yes, it is true to say that persistent fields are faster than FieldByName (and, although the test doesn't show it, the same is true for the other techniques for accessing fields). However, as with all performance tests, one must be both paranoid and also engage a little common sense. Simple maths shows that the access of a persistent field is very approximately 1.5 one-hundred-thousandths of a second faster on my PC. Given this difference, I have to ask, does anyone care?

➤ Figure 1



Drag And Drop

As I have already mentioned, the creation of persistent fields allows you to drag fields from the Fields Editor onto a form. This is a very impressive feature to demo, but in time its limitations become apparent. The problem is that the Fields Editor decides which components will be created for which fields. In many ways it makes very good guesses, given the components of which it is aware. For example, a string field will be given a TDBEdit, a memo field will be given a TDBMemo and a graphic field a TDBImage. All good choices, but date fields, integer fields and float fields are also given TDBEdits, which is not a good choice. One solution is to use Delphi's Data Dictionary. As each field is dragged from the Fields Editor and dropped onto the form, the Fields Editor reads the field's associated Attribute Set from the data dictionary and reads the Attribute Set's Control Class. The Data Dictionary allows you to enter a class name in this field specifically for this purpose. So if you go to the trouble of creating, say, a TDBDateTimePicker for date fields you can enter this in the associated field's Control Class and this control will be dropped onto the form.

However, this isn't such a great solution as it might first appear to be. The problem is that it requires the developer firstly to set up and use the Data Dictionary and then to enter the Control Class for every attribute set. If the intention is simply to change the default control class for a specific field type this is a lot of extra work to add to a project. Furthermore, this solution is only available for BDE datasets, which doesn't help if you use ADO.

A better solution is to replace the designer used by the Fields Editor. Let me start by offering my sincere thanks to Brian Long for supplying the information to get this solution to work. But for Brian's generosity, this probably would have ended up in the *Delphi Clinic*. This solution does only work for Delphi 5, though. Thanks to Delphi 5 including the code for many of its property editors, we can now see how most of the Fields Editor works. Figure 1 shows the designer class hierarchy.

TDSDesigner contains the key method GetControlClass. The solution is to override this method and make your own changes here. Unfortunately, this isn't as straightforward as one would like. The class we want to write is very simple and is shown in Listing 3.

However, this class won't compile, thanks to the author of BDEReg placing the all-important TBDEDesigner in the implementation part not the interface part. The pragmatic solution is to copy the complete TBDEDesigner class from BDEReg into a second unit, say BDEReg2, and rename the new TBDEDesigner to TBDEDesigner2. As a general rule, though, if ever you encounter a problem where block copying code in this way is the only solution, then it is usually an

```
var
  i : integer;
  Start, Elapsed  : TDateTime;
begin
  Start:=Now;
  for i:=1 to 200000 do begin
    Caption:=
      Table1First_Name.AsString;
    // Caption:=
    //   Table1.FieldByName(
    //   'First_Name').AsString;
  end;
  Elapsed:=Start-Now;
  Caption:=TimeToStr(Elapsed);
end;
```

➤ Listing 2

➤ Listing 3

```
uses BDEReg;
type
  TSpecialBDEDesigner = class(TBDEDesigner)
  public
    function GetControlClass(Field: TField): string; override;
  end;
function TSpecialBDEDesigner.GetControlClass(Field: TField): string;
begin
  if Field is TDateTimeField then
    Result := 'TDBDateTimePicker'    // TDBDateTimePicker is a hypothetical class
  else if Field is TIntegerField then
    Result := 'TDBSpinEdit'          // TDBSpinEdit is a hypothetical class
  else
    Result := inherited GetControlClass(Field)
end;
```

```
TSpecialBDEDesigner =
  class(TBDEDesigner);
TSpecialCDSDesigner =
  class(TCDSDesigner);
TSpecialDSDesigner =
  class(TDSDesigner);
```

➤ *Listing 4*

indication that something is wrong. In this case, either the author of BDEReg never considered that someone may want to inherit from TBDEDesigner, or they wanted to ensure no one ever would.

However, another problem with this solution is that it only works for TBDEDesigner. It ignores TCDSDesigner (used by TClientDataSet) and TDSDesigner (used by all other datasets including ADO Express and InterBase Express). TCDSDesigner suffers the same fate as TBDEDesigner, as the author of MIDREG.PAS also put TCDSDesigner in the implementation part. Fortunately, TDSDesigner is in the interface part, so we do not have to resort to such crude methods for a new designer for it. It is not a difficult problem to overcome: it simply requires another two classes which are identical to TBDEDesigner except for the class from which they inherit. Listing 4 shows the forward declarations of all three new class declarations.

A cleaner but less functional alternative to this solution would be to inherit from TDSDesigner instead of TBDEDesigner. Although this solves our problem of being able to drag and drop our own components, it results in the loss of all of the Data Dictionary features for BDE components (which matters if you use the Data Dictionary) and also in the loss of all TClientDataSet menu items (which will be unacceptable for some).

Having decided on which designers to write and how to write them, the next step in the solution is to tell Delphi to use our new designer(s). If you have ever created your own component editor you will know that in order to get Delphi to use your component editor you need to add a Register procedure which includes a call to RegisterComponentEditor. This is exactly what Delphi does to register its own component editors. Listing 5 shows various calls to RegisterComponentEditor extracted from various units in Delphi 5's property editor source code.

All of the component editors listed here inherit from TDataSetEditor, which includes a method called GetDSDesignerClass. It is this method which the Fields Editor calls to get important information about how to behave. In fact, the whole relationship between the dataset editors, the Fields Editor and the designers is a small plug-in system to allow parts of the editors to be reused. So, to get the Fields Editor to use our designer, we have to create a new dataset editor and register it in place of the old one. Listing 6 shows the class we want to write.

Unfortunately, we suffer the same problems with the dataset editor class as we did with the designer class: we don't have access to the classes we want to inherit from (eg TTableEditor) and we have to write a separate dataset editor for each component. If you don't want to copy the code for each of the inaccessible classes out then you can inherit from TDataSetEditor.

After all that, you can now drag and drop fields from the Fields Editor and get your own components dropped onto the form. This month's disk includes an Enhanced Fields Editor which reads the control classes from an INI file.

Resistance To Change

I've talked a lot about the benefits of persistent fields but I haven't said much about their disadvantages. That's what this section is about. Persistent fields resist change. They don't like the underlying database structure to change. This is unfortunate because the only constant which you can be sure of is change. Change is guaranteed to occur and database structures will always change. It used to be that the slightest little change to any field would present a problem if you used persistent fields. For example, if your form used persistent fields and you changed the size of a field from length 20 to length 25 then the next time you started Delphi and opened your form you would get an error telling you that the actual field size was different to that which the persistent field expected. Worse than this, the associated table would be closed without any indication that it was closed. If the project was then saved, the table would be saved in a closed state. This is one of the reasons why some programmers feel that all tables should be opened during the form's OnCreate event. Fortunately, as from Delphi 5, this behaviour has been changed. If a field's size changes the relevant persistent field no longer complains that it is expecting a different size and therefore it no longer surreptitiously closes the table. Thus we can welcome back those prodigal programmers and tell them they no longer have to open their tables in their OnCreate events.

However, all is not completely rosy, as persistent fields still complain if the data type changes. In many ways this behaviour should be expected and, even if an application didn't use persistent fields, it would still feel the shock of a change of data type of a field. However, it is a problem if you want to write applications which are portable across different databases. For most data types, Delphi manages a translation which doesn't cause a problem with persistent fields. However, a few data types do not translate well from one database format to another and these give persistent fields a problem. For example, Paradox BYTE fields will translate to InterBase

➤ *Listing 5*

```
RegisterComponentEditor(TTable, TTableEditor);
RegisterComponentEditor(TQuery, TQueryEditor);
RegisterComponentEditor(TStoredProc, TStoredProcEditor);
RegisterComponentEditor(TADODataSet, TADODataSetEditor);
RegisterComponentEditor(TClientDataSet, TClientDataSetEditor);
```

```
type
  TSpecialTableEditor =
   class(TTableEditor)
  protected
    function GetDSDesignerClass:
       TDSDesignerClass; override;
  end;
function TSpecialTableEditor.
  GetDSDesignerClass:
  TDSDesignerClass;
begin
  Result := TSpecialBDEDesigner
end;
procedure Register;
begin
  RegisterComponentEditor(
   TTable, TSpecialTableEditor);
end;
```

➤ *Listing 6*

VARCHARs. The persistent field type for a Paradox BYTE is a TBytesField and it gives a '*Type mismatch for field XYZ expecting Bytes actual String*' exception when used with the equivalent InterBase VARCHAR. However, it could also be argued (convincingly, in my opinion) that if portability is an issue then one should stick to data types which are known to be compatible across database formats and thus the problem goes away.

The last resistance to change you can expect to encounter when using persistent fields is the deletion of old fields and addition of new fields. If a field is deleted then when a form which uses a persistent field for the deleted field is opened you get a '*Table1: Field XYZ not found*' error. You have to delete the persistent field and then reopen the table. Now, if the form is used for editing and contains many controls on it where one refers directly to the deleted field, then this is a similar error to that which you would get even if you weren't using persistent fields (because you are trying to use a field which no longer exists). However, if the form simply displays a grid then it is a small nuisance because most grids simply display all available columns, whatever they may be. Similarly when a new field is added to the database you have to update the list of persistent fields before you can make use of the field.

Friend Or Foe?

All in all I have to say that the benefits of persistent fields far outweigh the disadvantages. The compile-time syntax checking is probably worth it in its own right, but the fact that you can't realistically use calculated, lookup or aggregate fields without them is also compelling. The performance issue is a red herring, because any performance benefit which is measured in hundredths of a millisecond is no benefit at all. The drag and drop has always been enticing but with the inclusion of the source code to the property editors in Delphi 5 it, at last, can be used to its full potential. Even the few drawbacks associated with persistent fields became fewer with Delphi 5's removal of the exception raised when the field size changes. All in all it is difficult to argue against using persistent fields.

Guy Smith-Ferrier (gsmithferrier@EnterpriseL.com) is Technical Director of Enterprise Logistics Ltd (www.EnterpriseL.com), a training company specialising in Delphi, which is now running ADO courses.