SQL Components

A framework for building search forms against relational databases

by Nando Dessena

Those among us who work on database applications with a client/server architecture have undoubtedly had the chance to notice one fact: the standard Delphi VCL, which offers many facilities to handle local databases, does not give us the best possible performance when working with SQL database servers. It doesn't take an expert to notice that searching and filtering techniques which are best suited for Paradox tables (such as `Locate`, `SetRange` and `Filter`) will not behave as smoothly when they are converted into the only language that an RDBMS understands: SQL.

So, the idea upon which this article is based is to create some components for searching and filtering data that are specifically designed for the SQL language. Furthermore, I'm going to discuss a possible framework that will make extending the components and creating new ones a simpler task. What I'm going to describe is nothing more than a start, and will need various enhancements and refinements before it can be used to cover all the needs of a real-world application. Nevertheless, I hope the concepts and techniques will prove useful and interesting.

Balancing The Workload

We know that, in a client/server environment, it is best to arrange things so that the server does most of the work. Having the client perform lookups, table joins and, most of all, filtering and sorting the data, is a recipe for disaster. The only circumstance in which it is acceptable to perform this kind of processing client-side is when the data is already there for some other reason, for example with the so-called 'briefcase model'. Fetching all the data from the server just to filter or sort it, you will agree, will make us high users of network bandwidth and cause delays for our users. So, the best way to do these things is through the SQL language.

Although a large portion of the Delphi VCL is dedicated to databases, it doesn't help us very much: we can see that `Lookups`, `Ranges` and `Filters` are not suited to a client/server architecture. Since the database VCL was born when the BDE was still in common use, it shares its flattening philosophy, which says all databases should be treated almost the same way. Compromises rarely represent the best option for anyone, in this case those of us who do client/server development. As a consequence, I am going to introduce some simple examples of how you can leverage the power of SQL without giving up on a hierarchy of reusable classes. An SQL statement is surely the best way to query a relational database, but it is not mandatory to write this statement by hand each time!

Filtering Components

To build our family of components (each of which will be named after its base class plus an `SQL` prefix, adding also an extra `DM` just to be sure we avoid any clashes), let's start from a `TQuery` descendant (`TDMSQLQuery`) which will be capable of modifying its `where` clause upon request. Such a request could be made by calling a method and passing a set of controls, each of which represents a filtering criterion (such as a `TDMSQLEdit`, a `TDMSQLListBox`, and so on). By putting a bunch of these controls (and perhaps a `DBGrid`) on a form we can have an effective SQL-based filtering system. For example, suppose we have a couple of `TDMSQLEdit` components as defined in Listing 1, and a `TDMSQLQuery` with the following SQL statement:

```
select Code, Description
  from Goods
```

By executing this line of code:

```
MDSQLQuery1.BuildSQL(
  [SQLEdit1, SQLEdit2])
```

we automatically transform the 'basic' SQL statement in `MDSQLQuery1` into:

```
select Code, Description
  from Goods
  where Description like
  "Metal%" and Status = "A"
```

To implement the system we will need a set of routines to extract parts of an SQL statement (`select`, `from`, `where`, `order by` and so on). Since the structure of SQL sentences is usually quite simple, we won't need a real SQL parser (which you can find on the internet in freeware implementations, by the way), that would go beyond the purpose of this article. We will put together a set of simple routines based on `Pos` and `CompareText`. These routines are contained in the `SQLUtils` unit. After that, we need a way for the components to 'know' each other but without coupling them too tightly. We have two kinds of components here: queries (we don't want to limit ourselves to `TQuery` descendants, but would like to be able to define `TDMSQLIBQuery`, `TDMSQLADOQuery` and such with ease) and criteria. A criterion can be an `Edit` component, as in the previous example, but also a `ListBox`, a `RadioGroup`, a `DateTimePicker`, and so on.

Criteria have some common characteristics and behaviour, but it is clear that they cannot inherit

➤ *Listing 1*

```
object DMSQLEdit1: TDMSQLEdit
  DataField = 'Description'
  SQLOperator = opBeginningWith
  Text = 'Metal'
end
object
  DMSQLEdit2: TDMSQLEdit
  DataField = 'Status'
  SQLOperator = opEqualTo
  Text = 'A'
end
```

from a common base class. Since Delphi does not have multiple inheritance (for which many are always grateful to Borland), I have decided to use a special interface to group my criteria. Thus we introduce at this point an interface called `IDMSQLCriterion` and state that every criterion must implement it. For this reason, we should keep the interface as light as possible, since we intend to be able to define new criteria easily.

For now, we only need two methods. First, `GetSQL` returns the SQL string corresponding to the criterion (the `DMSQLEdit1` component in our example would return the string `Description like "Metal%"`, maybe with additional bounding parentheses), while `ClearSQL` empties the criterion (in our case, the implementation would just call `TEdit.Clear`).

A query component must put together all the SQL snippets from the criteria using the logical operators `and` and `or`, depending on the situation. We introduce the `IDMSQLQuery` interface for this purpose, since we cannot find a common base class for all the possible query components we plan to define.

Architecture

Implementing the basic capabilities for these two interfaces requires us to write a respectable amount of code. So, I have decided to encapsulate the real code into two internal classes. The first is `TDMSQLCustomCriterion`, which may have some derived classes, to which a criterion control delegates the implementation of `IDMSQLCriterion`. The second is `TDMSQLQueryImpl`, which includes all the capabilities each query object must have. This kind of encapsulation will prove useful when we create new components: it will suffice to instantiate an internal helper object to have the appropriate interface implemented, thanks to the interface implementation delegation which was introduced in Delphi 4.

These internal objects, in turn, need to get information from their owners in order to work correctly.

```
// Every criterion component must implement this interface.
IDMSQLCriterion = interface
  ...
  // Returns the complete SQL expression relative to the criterion.
  function GetSQL: string;
  // Clears the criterion.
  procedure ClearSQL;
end;
// Used by a TDMSQLCriterion object to read information from the
// owning object.
IDMSQLCriterionData = interface
  ...
  // Returns the value(s) of the criterion; Unassigned means no chosen
  // criterion.
  function GetSQLValue: Variant;
end;
```

➤ *Listing 2*

For example, an object of class `TDMSQLCustomCriterion` needs to know the current value (or values, in the case of a multi-select criterion) of the owning control, in order to be able to build the SQL snippet. So we introduce another interface, `IDMSQLCriterionData`, that all criteria must implement (this time directly, without any delegation). This last interface features a method called `GetSQLValue` which returns a `Variant`; in the case of a multi-select criterion, the method will return an array of `Variants`.

In a similar manner, a query object implements the `IDMSQLQuery` interface with the fundamental method `SetSQLText`, which will help a contained `TDMSQLQueryImpl` object in substituting the SQL string in the container with the one obtained from the criteria. With this architecture, we can concentrate most of the code into two classes. Since we have kept the interfaces to a minimum, each new criterion we want to define will just have to implement two interfaces and a total of three methods. Listing 2 shows an excerpt of the interface definitions from the `DMSQLBase` unit.

This kind of bidirectional communication between owning and owned objects is made feasible thanks to the interfaces. It is a very powerful technique which is rarely used in real-world applications, maybe simply because not all Delphi programmers know how to exploit the features of interfaces and interface implementation delegation.

The 'abstract' class `TDMSQLCustomCriterion`, just like the main interface it implements, has one method that stands out from the others: `GetSQL`. This method makes use of several other private and protected methods that query the contained object through the `IDMSQLCriterionData` interface and build the final `where` clause portion. Most of the work is done in the abstract method `BuildSQL`, that derived classes must implement. For now, we have two derived classes: `TDMSQLSingleCriterion` for single-value criteria (such as an `Edit` control) and `TDMSQLMultipleCriterion` for multi-select criteria (such as a multi-select `ListBox`).

Once we sort out who is in charge of what, and write most of the code in the `DMSQLBase` and `DMSQLUtils` units, we need to create the actual criteria; all we need to do is inherit a new class from `TEdit` (or whatever `TSuperDuperEdit` class we need to use), implement the relevant interfaces and publish some useful properties. In our example, I chose to cut the corners and publish the `TDMSQLSingleCriterion` object itself as a property, so that its sub-properties are directly seen and manipulated in the Object Inspector, and streamed from or to the .dfm file. This is possible because we have been smart and inherited `TDMSQLCustomCriterion` from `TPersistent`. Table 1 shows a list of the main properties of a criterion internal object, together with a brief explanation of the meaning of each one.

A criterion could also surface, one by one, the properties of its contained object; this gives us a little more flexibility at the price of more lines of code. By the way, we would also win a free property editor for the `DataField` property, since the unknown author in

```
type
  // An Edit box criterion.
  TDMSQLEdit = class(TEdit, IDMSQLCriterion,
    IDMSQLCriterionData)
  private
    FCriterion: TDMSQLSingleCriterion;
    function GetSQLValue: Variant;
    procedure ClearSQL;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Criterion: TDMSQLSingleCriterion read
      FCriterion write FCriterion implements IDMSQLCriterion;
  end;
...
constructor TDMSQLEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FCriterion := TDMSQLSingleCriterion.Create(Self);
end;
destructor TDMSQLEdit.Destroy;
begin
  FCriterion.Free;
  inherited;
end;
function TDMSQLEdit.GetSQLValue: Variant;
begin
  Result := SQLStringToVariant(Text);
end;
procedure TDMSQLEdit.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  FCriterion.CustomNotification(Acomponent, Operation);
end;
procedure TDMSQLEdit.ClearSQL;
begin
  Clear;
end;
```

➤ *Listing 3*

Borland decided to make the default editor available only if you inherit from `TComponent` (while we have a mere `TPersistent` at hand). If and when internal `TComponents` will become expandable in the Object Inspector, we will have a comprehensive solution in inheriting our contained object from `TComponent`. Anyway, Listing 3 has the code for the `Edit` criterion.

As you can see, the interface and implementation parts total a few lines of code, most of which are straightforward. The 'real' code is in the base units. `GetSQLValue` just returns the value of the `Text` property, handling the special cases of unassigned and empty criteria through `SQLStringToVariant`, from the `DMSQLUtils` unit (see Listing 4).

`Null` and `Unassigned` are treated as special values when it comes to building the SQL string. `NullValueStr` is defined as `<null>` and it is what the user will have to type in when he wants to search for `NULL` values. Having said this, building more criteria is not a complex task. As an example, I have built a `ListBox` criterion that supports multiple selection (`value = "this" or value = "that"`). The parts in

➤ *Listing 4*

```
function SQLStringToVariant(
  SQL: string): Variant;
begin
  if SQL = NullValueStr then
    Result := Null
  else if SQL = '' then
    Result := Unassigned
  else
    Result := SQL;
end;
```

Property Name	Description
DataSource	Together with `DataField`, this property serves to link to a field in the DataSet to filter. If persistent fields are defined, and if the `SQLOptions` property does not include the `sqlOverrideFieldType` flag, this link is followed to obtain the data type, otherwise the value of the `DataType` property is used. Automatic reading of the data type works only with persistent fields because it has to be done while the query is not open.
DataField	The name of the field on which this criterion is applying the filter; this is required.
DataType	The data type must be specified when it is not possible to determine it automatically (see also `DataSource`).
SQLOperator	The SQL operator to use for applying the filter (equal to, not equal, greater than, etc.); it is required.
SQLOptions	The meaning of the `sqlOverrideFieldType` is explained before; `sqlAddBrackets` ensures that the resulting string is bound with round brackets (to avoid ambiguities in concatenating the criteria).

➤ *Table 1*

➤ *Listing 5*

```
function TDMSQLListBox.GetSQLValue: Variant;
var
  VarIndex: Integer;
  g: Integer;
begin
  if not MultiSelect then begin
    Result := VarArrayCreate([1, 1], varVariant);
    // Single selection: return a single value.
    if ItemIndex < 0 then
      Result[1] := SQLStringToVariant('')
    else
      Result[1] := SQLStringToVariant(Items[ItemIndex]);
  end else begin
    // Multiple selection: return an array.
    Result := VarArrayCreate([1, SelCount], varVariant);
    VarIndex := 1;
    for g := 0 to Pred(Items.Count) do
      if Selected[g] then begin
        Result[VarIndex] := SQLStringToVariant(Items[g]);
        Inc(VarIndex);
      end;
  end;
end;
procedure TDMSQLListBox.ClearSQL;
var
  g: Integer;
begin
  if MultiSelect then begin
    for g := 0 to Pred(Items.Count) do
      if Selected[g] then
        Selected[g] := False;
  end else
    ItemIndex := -1;
end;
```

which the code differs from that of the `Edit` criterion are the `GetSQLValue` and `ClearSQL` methods, which you can find in Listing 5.

On to the query object. The first implementation is a class derived from `TQuery`, but we can use a non-BDE query object as well (the code on the disk has a `TDMSQLIBQuery` installed in a separate package, so that we have the basic package, the BDE package and the IBX package for InterBase Express). As you might remember, a 'searchable' query is identified by the `IDMSQLQuery` interface (see Listing 6, from the `DMSQLBase` unit).

The most interesting part in the declaration of the query object, which implements this interface, is shown in Listing 7.

The internal `TDMSQLQueryImpl` is directly surfaced as a published property, so that it can be seen and manipulated from the outside in general and the Object Inspector in particular (just as we did with the criteria). The `TDMSQLQueryImpl` features everything that is needed to make `TDMSQLQuery`, `TDMSQLIBQuery` and such work; look at its public interface in Listing 8.

`BaseSQL` is the SQL string to which the criteria are added when you call `BuildSQL`. This allows us to have fixed criteria (by adding a `where` clause to the base string). As a consequence, we must remember to take care of the value of this property in case we need to modify the SQL string at runtime. The component, in fact, will only assign it to the value of the SQL property when it is streamed from the .dfm file (in the `Loaded` method, that is).

The `BuildSQL` method, in its three incarnations, creates a new SQL statement based on the input criteria. We can tell the method which criteria we intend to use in three ways (directly, by owner or by callback), for maximum flexibility. Three overloaded versions of the `ClearCriteria` method exist as well (they all make use of the `ClearSQL` method of the `IDMSQLCriterion` interface).

Finally, `SQLConnector` indicates the logical operator to be used to connect the criteria (`or`, `and`), while the `SQLOptions` are for customising the creation of the SQL statement. The only available option, for now, is `sqlOpenAfterBuild`, which means the query will be automatically opened after a call to `BuildSQL`.

```
// Identifies a query object in our framework.
IDMSQLQuery = interface
  ...
  // These two methods are called by the TDMSQLQueryImpl object before and
  // after each call to BuildSQL.
  procedure BeforeBuild(Sender: TDMSQLQueryImpl);
  procedure AfterBuild(Sender: TDMSQLQueryImpl);
  // This method is called by the TDMSQLQueryImpl after a BuildSQL (and before
  // AfterBuild) to set the newly built SQL statement. Usually, a query object
  // will assign the value to its SQL property.
  procedure SetSQLText(Value: string);
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
// A TQuery that can change its where clause.
TDMSQLQuery = class(TQuery, IDMSQLQuery)
private
  FImplementor: TDMSQLQueryImpl;
  procedure AfterBuild(Sender: TDMSQLQueryImpl);
  procedure BeforeBuild(Sender: TDMSQLQueryImpl);
  procedure SetSQLText(Value: string);
  ...
published
  property Implementor: TDMSQLQueryImpl read FImplementor write FImplementor;
end;
```

```
TDMSQLQueryImpl = class(TPersistent)
public
  // The base SQL statement; this property must be assigned a value at least
  // once by the query object (f. ex. in the Loaded method).
  property BaseSQL: TStrings read FBaseSQL write SetBaseSQL;
  // Methods to build the SQL string from:
  // - an array of criteria.
  procedure BuildSQL(Criteria: array of IDMSQLCriterion); overload;
  // - all the criteria owned by AOwner.
  procedure BuildSQL(AOwner: TComponent); overload;
  // - all the criteria passed by the callback function.
  procedure BuildSQL(EnumProc: TDMSQLCriteriaEnumProc); overload;
  // Methods to clear the criteria.
  procedure ClearCriteria(Criteria: array of IDMSQLCriterion); overload;
  procedure ClearCriteria(AOwner: TComponent); overload;
  procedure ClearCriteria(EnumProc: TDMSQLCriteriaEnumProc); overload;
published
  property SQLOptions: TSQLOptions read GetSQLOptions write FSQLOptions
    default SQLOptionsDefault;
  property SQLConnector: TSQLConnector read FSQLConnector
    write SetSQLConnector default SQLConnectorDefault;
end;
```

➤ *Listing 8*

Putting It All To Work

To show how these components work, I have developed an example called SQLDemo (SQLDemo_IB is the IBX version). The program consists of a search form to filter data contained in the `EMPLOYEE` table of the `IBLOCAL` sample InterBase database. The main idea behind this example is that, since we have encapsulated the most commonly needed features in the components, it is possible to create a simple search form with a bunch of components and almost no code. Figure 1 shows the search form at runtime: it has a series of filter criteria in the upper left part of the form, a `DBGrid` towards the bottom showing the filtered data, and a `Memo` (which is there only for demonstration purpose) showing the generated SQL statement. You can generate a new statement with the `Refresh` button, and empty the search criteria with the `Clear` button. A `TDMSQLQuery`, a `TDataSource` and a `TDatabase` complete the picture.

Each criterion is linked to the `DataSource`, and features the properties we described earlier. See Listing 9 for an example.

The code for the `Refresh` button is straightforward, and is shown in Listing 10.

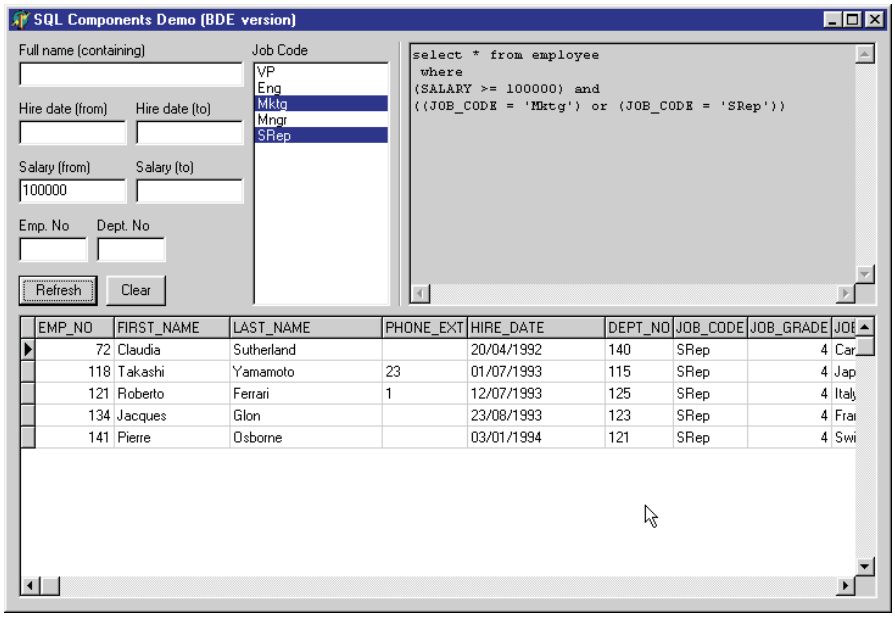
First we ask `QyResult` to rebuild its SQL instruction from the criteria on the form, then we show the resulting string in the `Memo` (the `try..finally` block is there to show the SQL string even in the case of errors, mainly for debugging purposes). We have set the `sqlOpenAfterBuild` option so we don't need to manually open the

query. The `Clear` button just calls the query's `ClearCriteria` method. Except for a line of code to open the `Database` component, we don't need anything else to implement our fully functional and efficient search form against a database table.

The IBX demo program is almost the same: it needs an additional component (a `TIBTransaction`) plus IBX versions of the data access components, but the code itself is identical. The same applies for the code of the `TDMSQLQuery` and `TDMSQLIBQuery` components, with almost no differences. This kind of architecture allows us a certain level of abstraction from the middleware, and I bet that it will take only a few minutes to inherit a search query from a dbExpress query (whatever they will call it) once it (together with Kylix or Delphi 6) is out.

Possible Extensions

To extend the framework, the first thing that comes to mind is the introduction of new criteria. I have included only `ListBox` and `Edit` criteria, but it would be easy to create some additional ones: `TDMSQLDateTimePicker`, to handle date and time fields, `TDMSQLCheckBox`, for `Boolean` fields (which must also be added to the set of supported field types), `TDMSQLComboBox` (similar to `TMDSQLListBox`), `TDMSQLLookupListBox` (a `TDMSQLListBox` bound to a `DataSet`), `TDMSQLLookupComboBox` (a `TDMSQLComboBox` bound to a `DataSet`), `TDMSQLRadioGroup` (similar to `TDMSQLListBox`), `TDMSQLLookupRadioGroup` (a `TDMSQLRadioGroup` bound to a `DataSet`). I'll leave the rest to your imagination!



➤ *Figure 1*

If you use third-party components to build your user interfaces, then you could build new criteria based on such components.

Anyway, it is possible to extend the framework itself, perhaps getting rid of some of the limitations I have introduced for simplicity. Parameterised queries are not supported; you can't filter `group by` queries on the `having` clause; you can use senseless operators, such as `opBeginningWith` with a date field.

With regard to adding functionality, we are only limited by our own fantasies. We can have a `ComboBox` linked to a criterion, used to set its `SQLOperator` property. We could have a `RadioGroup` or `ListBox` with the same function, a `ComboBox` or `ListBox` to set the value of the `DataField` property. We can even have a system to write a set of criteria to a stream or a file and read it back; this could be useful to save and reload user queries, or to generate report headers, it could also be a first step for the creation of a two-way system.

Conclusion

The main concept behind this article is that it is not always necessary to give up a nice GUI or easy RAD techniques to gain efficiency. This said, there are no limits to the extent of encapsulation of the SQL language into reusable classes and visual components. As we have seen, Delphi comes to help us with the VCL, interfaces and a rich data access layer. The weak part of the equation, that is the visual controls for searching and filtering that are inefficient in a client/server environment, can be substituted with a custom solution as we have done in our example. By paying enough attention to the architectural aspects of the framework, we can leverage some language features like interfaces and interface implementation delegation to make it easy to use and to expand.

```
object EtFullName: TDMSQLEdit
  ...
  Criterion.DataSource = DsResult
  Criterion.DataField = 'FULL_NAME'
  Criterion.SQLOperator = opContaining
end
```

➤ *Above: Listing 9*

➤ *Below: Listing 10*

```
procedure TFmMain.PbBuildSQLClick(Sender: TObject);
begin
  try
    QyResult.Implementor.BuildSQL(Self);
  finally
    EtSQL.Lines := QyResult.SQL;
  end;
end;
```

Nando Dessena works as a Delphi developer, trainer, consultant and technical author in Italy (not necessarily in this order). You can reach him by email at nandod@dedonline.com