Surviving Client/Server: Interactive Query Building

by Steve Troxell

As databases increase in size and complexity, so do the demands put on them by users. Try as we might to build the most feature complete applications we can, users always seem to want a little bit more. By designing our applications to be data-driven wherever possible, we can allow users some degree of flexibility in making the system do things their way. The specific case I want to talk about this month is interactive query building.

When dealing with huge masses of data, many times users need to do specific things to specific groups of data. Many times these groups are hard to predefine in static forms in an application. For example, the marketing staff may be pulling many different subsets of customers from their database throughout the year to target for solicitation for new business. Let's get all the people who bought Super-Widgets within the last six months and send them a mailing about the new Mega-Widgets we've just come out with. Let's find everyone who subscribes to *Coding Commando* magazine and see if they'll buy the fresh new book *Rapidly Coding Complete Development*.

Likewise, for a business with a large number of employees, there may be many uses for custom subsets for purposes of benefits, payroll, project management, etc.

SQL is a great tool for getting to data in the system without relying on having a programmer around to construct a specific screen in a specific application in advance. In fact, SQL was conceived as a tool to be used by information specialists to increase data accessibility without the need to tie up a skilled programmer. SQL requires a bit of skill itself, but not nearly the same skill as writing a Delphi or C++ program to extract data. Even so, SQL in the raw may still be a little demanding for some segment of your user base.

If we want to give our users the ability to define custom subsets of data we could ask them to supply syntactically correct SQL clauses which we could dynamically assemble into full queries and execute against the database. This approach would require the users not only to understand SQL syntax and operations, but also to know the field names and encoding schemes in our database.

As an alternative, there are commercially available 'visual query builder' components which we could plug into our applications to allow users to build SQL clauses by selecting field names and operations from pick lists. While this relieves the need to understand SQL directly, it still requires technical knowledge of the table structure and data.

Almost all database applications employ code tables, or lookup tables, to contain lists of related data codes and their descriptions. For example, to identify a temporary employee on a form, most applications will not require users to enter the code `TMP` in an `Employee Type` edit box. Instead, they will probably provide a dropdown list and allow the user to select the entry `Temporary` from the list of all possible employee types. This dropdown list would be populated from the code table for employee type. The application would translate the clear text description `Temporary` to the code value `TMP` for storage in the database. Throughout the application, the user only sees the clear text description and is most likely totally unaware of the encoding behind the scenes.

If we are making data entry this easy for users, let's use the same concept to simplify how we can let users define their own data subsets as well. Our users are already very familiar with the range of values available to them for the various data fields of interest. So why not provide them with a query building interface that leverages what they already know instead of demanding that they learn new skills?

A Bit Of Terminology

Before we proceed, it would be helpful to get a clear understanding of the terminology I'll be using to describe this query building system. The purpose of this system is to define a subset of data: specifically, in our example, a subset of employees. Since we are not operating on the entire population of employees, we call this a *qualified subset* of employees. We call the conditions that define the subset the *qualifiers*. For example, a subset may be defined by the single qualifier 'hourly employee'; another subset may be defined by multiple qualifiers 'salaried, regular employees in the accounting department.'

Each individual piece of data that is eligible to make up a qualifier is called a *filter field*. In the example used above, the data field defining whether an employee is hourly or salaried is a filter field, eligible for use in any set of qualifiers.

The distinction between filter fields and qualifiers is that filter fields are simply those data fields that we are allowed to use to define subsets of data. This is typically a static list. Qualifiers are what we construct with the filter field building blocks to define subsets of data. There may be many different sets of qualifiers, each defining a different cross-section of data, and each assembled from the same unchanging pool of filter fields.

System Architecture

There are several critical elements in the architecture for the system we will construct here. In many ways this is an oversimplified architecture to make it easier to explain the concept. At the end of the article we will discuss ways to extend the system for more powerful uses.

There are three requirements of our system. First, all filter fields must be contained within a single table. No multi-table joins are allowed in this simplified system. Second, all filter fields must contain discrete data values defined though some form of lookup table. Third, all filter fields must be predefined in a static table.

Obviously, these conditions dictate that the user will not have the full range of data fields available to them when defining a subset. This limitation can be taken as the cost of the ease of use of the system. In most cases, users will not *need* the full range of data fields and it would be fairly straightforward to provide them with a well-rounded list of candidate filter fields. We will also see that it is easy to add additional filter fields at a later time without recoding the system.

For our example, we will allow users to select from the Employees base table, using any combination of three filter fields: `empType`, `empSalaryOrHourly`, and `empFullOrPartTime`. The code values for these fields are stored in a single lookup table called `SystemCodes` as shown in Listing 1. Back in the May 1998 issue, we discussed this technique of consolidating multiple logical code tables into a single physical database table. We can still use this query-building system if our code tables are broken up into separate database tables: it's just much easier this way. Also, note that the code table scheme we talked about in the May issue allowed for a mixture of consolidated and standalone code tables, all accessed in one systematic manner.

```
SystemCodes
codTable            codCode     codDesc
EMPTYPE             CON         Contractor/consultant
EMPTYPE             INT         Intern
EMPTYPE             REG         Regular
EMPTYPE             STU         Student
EMPTYPE             SUM         Summer help
EMPTYPE             TMP         Temporary
EMPTYPE             Z           <None>
FULLORPARTTIME      F           Full time
FULLORPARTTIME      P           Part time
SALARYORHOURLY      H           Hourly
SALARYORHOURLY      S           Salaried
```

➤ *Listing 1*

```
QualifierFilters
qlfID qlfDescription        qlfCodeTable     qlfDataField

1     Hourly/salaried       SALARYORHOURLY   empSalaryOrHourly
2     Full time/part time   FULLORPARTTIME   empFullOrPartTime
3     Employee type         EMPTYPE          empType
```

➤ *Listing 2*

We set up our list of available filter fields in a table called `QualifierFilters` as shown in Listing 2. In this table we provide a unique identifier for each filter field, a description which we will provide in the user interface, a reference to the applicable code table, and an association with the actual data field in the table definition.

To this point we've defined all the data necessary to present our query-building user interface. The only thing left to do is provide a place for users to store their qualifier definitions. Listing 3 shows the structure for the `Qualifiers` table (for MS SQL Server).

The `Qualifiers` table contains all the qualifier sets we've defined. Each set is identified by the `quaID` field and there may be one or more rows in each qualifier set. Each row in the table is uniquely identified by the `quaRecID` column, which is simply an autoincrementing field starting at 1000. We started at 1000 simply for illustration purposes to help distinguish its values from the `quaFilterID` field, which is a link to the `QualifierFilters` table defining which filter field we are using. Finally, `quaValue` is the actual code value we will associate with this particular filter field.

For example, if we wanted a set of all salaried, full-time employees, Listing 4 shows the values we would have in `Qualifiers`. If we relate the `quaFilterID` column back to the `QualifierFilters` table, we see that a value of `1` ties to the `SalaryOrHourly` code table and a value of `S` means salaried. Also, a `quaFilterID` of `2` ties to the `FullOrPartTime` code table and a value of `F` means full-time. The `quaID` field is just arbitrarily assigned by the system to distinguish this set of qualifiers from other sets we might define.

You might think that the `quaRecID` autoincrement field is unnecessary because `quaID` and `quaFilterID` is enough to uniquely identify any row in `Qualifiers`. This would be true if we prevented users from specifying more than one value for any given filter field. For example, suppose we wanted a set of all student and summer help employees. Listing 5 shows the values that we would have in `Qualifiers`, and you can see we would have duplication in the `quaFilterID` column.

From the examples shown in Listings 4 and 5, it should be evident that our query-building engine will assume a logical `AND`

➤ *Listing 3*

```
CREATE TABLE Qualifiers(
  quaRecID int IDENTITY(1000,1),
  quaID smallint,
  quaFilterID smallint,
  quaValue varchar(20),
  PRIMARY KEY (quaRecID)
)
```

➤ *Listing 4*

```
Qualifiers (example values)
quaRecID quaID quaFilterID quaValue
1000     100   1           S
1001     100   2           F
```

```
Qualifiers (example values)

quaRecID  quaID  quaFilterID  quaValue
1000      100   3            STU
1001      100   3            SUM
```

➤ *Listing 5*

```
SELECT quaRecID, quaID, quaCode, quaFilterID, qlfCodeTable
  FROM Qualifiers, QualifierFilters
  WHERE quaFilterID = qlfID AND
        quaID = :quaID
```

➤ *Listing 6*

```
SELECT qlfID, qlfDescription, qlfCodeTable
  FROM QualifierFilters
  ORDER BY qlfDescription
```

➤ *Listing 7*

between different filter fields and a logical `OR` between multiple references to the same filter field.

The Front End

The are two sides of the query building process: the front end, where, the user defines their qualifiers, and the back end where that definition is turned into workable logic to manipulate the database. While the back end is where all the action will take place, there are a few challenges in implementing the front end, so let's take a look at that first.

Some form of grid is called for here since the user will be selecting one or more filter field entries to construct their qualifiers. In concept we have a two column grid where the cells in the first column have a dropdown list of all the available filter fields and the cells of the second column have a dropdown list of all the code values available *for that particular filter field* (see Figure 1). This is the main challenge of the front end. Since each row can represent a different filter field with a different set of code values, the contents of the dropdown list for the second column can vary from row to row. Delphi's `TDBGrid` was not designed with this possibility in mind, but we will see how we can make it do our bidding without resorting to a custom descendant component. Most likely you will have your own in-house `TDBGrid` descendant or a third-party grid control, so your specific implementation obviously will vary.

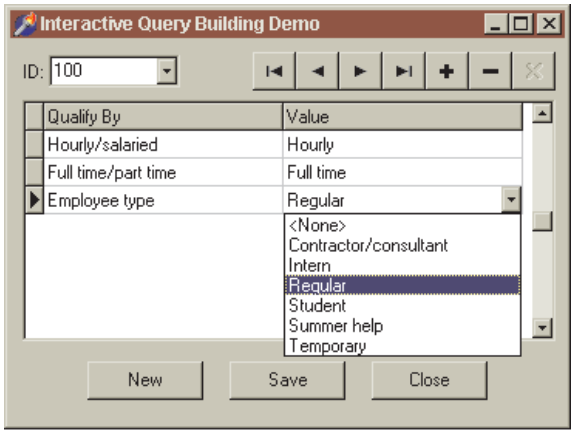
The main query (the one the `TDBGrid` is bound to) is based on the `Qualifiers` table of course (see Listing 6). We join to the `QualifierFilters` table in order to pick up the code table association for that particular qualifier. The columns in the grid are defined through the `TDBGrid.Columns` property, which automatically provides us with a dropdown list edit control if we either bind the column to a lookup field in the dataset, or provide a list of strings ourselves to serve as the dropdown list contents.

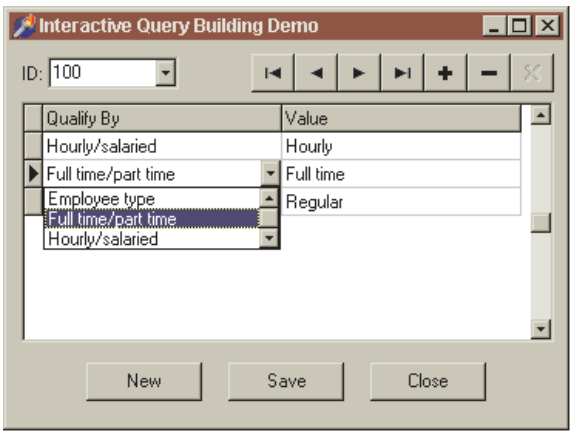
The first column in the grid is bound to the `quaFilterID` field but displays the filter name. This is a simple lookup field definition using a secondary query to fetch the rows from the `QualifierFilters` table (Listing 7), associate `Qualifiers.quaFilterID` with `QualifierFilters.qlfID`, and return `QualifierFilters.qlfDescription` as the lookup value. There is nothing really special about the setup of column one. The dropdown list is provided for us by virtue of being bound to a lookup field. This suits us fine for this column since our dropdown list contents will remain static as we move from row to row (see Figure 2).

The second column is more problematic since we must change the dropdown list contents depending on which filter field the row represents. Since we are storing the code *value* and displaying the code *description*, we naturally tend to think of using a lookup field. We might be tempted to create a lookup query like this:

```
SELECT codCode, codDesc
  FROM SystemCodes
  WHERE codTable = :qlfCodeTable
```

We would use a parameterized query here tied to the main dataset in a master-detail relationship so that as we moved from row to row in the main dataset, we would always have the appropriate code table dataset for that filter field. Then we would simply create a





➤ *Top: Figure 1*

➤ *Bottom: Figure 2*

lookup field binding quaCode to the codCode and returning the codDesc as the code description.

This approach does indeed work fine as far as keeping the actual dropdown list in sync with the proper values for any given row. But since the contents of the lookup dataset change when we move from row to row, the VCL cannot properly maintain the values in other rows, since their data values do not always correspond to what is currently available in the lookup dataset. Furthermore, as we moved from row to row we would constantly be firing off queries to the server to populate the lookup dataset. A better design would attempt to eliminate extraneous queries resulting from simple user interface navigation.

So what do we do? One way or another we have to emulate a lookup field in code. To avoid extraneous queries to fetch the code table values every time we moved among the rows in Qualifiers, I chose to implement a code table cache. The QualifierFilters table identifies all possible code tables that might be needed, so we simply load all possible code values and their descriptions internally into TStringLists and use code to populate the grid cells and dropdown lists as needed.

I'll describe the code table cache briefly: you can get full details of its implementation from the code supplied on this month's disk. The cache consists of a single TStringList containing the names of all the code tables identified in QualifierFilters. For each code table entry in the string list, the Objects property refers to another TStringList containing all the code values and descriptions for that code table (in <value>=<description> format).

To set up our own dropdown list for a grid cell, we use the TDBGrid.Columns.PickList property. This property holds the TStringList to use for the dropdown list for all cells in the column. Since there is one pick list per column, we need to clear it and repopulate it as we move between rows in the dataset. We can easily do that with the AfterScroll event handler.

The second column of our grid is bound to the quaCode field. This holds the code value, and we want to display the code description. Since we have all the code values and descriptions cached in memory, we can add an OnGetText handler for this field to lookup the code value in the cache and return its description. Likewise, when we pick a new code from the dropdown list, we are really picking its description and writing that to the quaCode field. So we also need to provide an OnSetText handler to do the reverse by looking up the code description in the cache and returning its matching value.

Voila! We now have a lookup column in a grid whose dropdown list contents vary from row to row.

The Back End

The whole point of this endeavor is to take the information the user has given us and formulate an SQL query out of it. More specifically, we need to formulate an SQL query WHERE clause. Figure 3 shows an example set of qualifiers. This corresponds to data in the Qualifiers table as shown in Listing 8. From this we want to obtain the SQL query shown in Listing 9.

Obviously, we'll need a query to give us a dataset like that shown in Listing 8, but we'll also want to join to the QualifierFilters table in order to include the qlfDataField column which tells us which filter fieldname to use (refer back to Listing 2). Then we simply loop through this dataset and for each row, we transform the values in qlfDataField and quaCode into an expression for the WHERE clause. The expressions are connected by AND so we get all rows that match all our qualifiers.

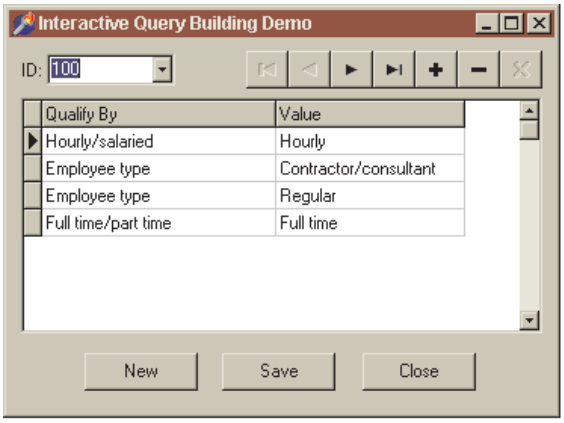
As we are stepping through the rows in Qualifiers, we check for duplication in the quaFilterID field. As long as there is duplication, we build a 'sublist' of expressions for that particular filter field, concatenated by ORs. When we've got them all, we encase the whole thing in parentheses and use that as the next 'single' expression in our WHERE clause. The coding for this is actually fairly straightforward and an example of it is included in the demo program on the disk.

Taking It Further

The system I've described here has been scaled down and has severe limitations as it stands for a database of any complexity. Obviously, the physical code tables shown here are oversimplified, but even if we have more complex code tables, we only need a value and a description here. So using the scheme we discussed back in May, there still wouldn't be too much deviation from what I've laid out here.

We should also open up the candidates for filter fields by allowing field values that aren't necessarily expressed by code tables. For example, we may want to select records based on postal code (zip code in the US). We obviously don't want to pick this from a dropdown list, but would rather key it in directly. We may even want to select records based on date fields, or even a range of dates. We can handle this by extending the QualifierFilters table to identify different classes of data field. Then, just like we currently change the contents of the dropdown list based on which filter field we are on, we might change the cell editor control altogether based on which filter field we are on. For date fields we might perhaps pop up a calendar control and enable a third column in the grid so that we can specify a range of dates for a single filter field.

The concept of user-defined qualifiers can be used for other functions as well. Users could also select fields to update with new values and let the qualifiers define the range of records that will receive this 'mass update'. We can also use this information when entering a new employee, customer, or whatever, to default certain fields to specific values depending on what 'profile' the new entity matches.



➤ *Figure 3*

Conclusion

When defining selection criteria, users will think in terms such as 'I want to deal with all the part time, hourly employees', not 'I want to deal with all the employees where `StaffType = 'P'` and `WageType = 'H''`. All things being equal, any user interface that operates closer to how the user thinks, and makes fewer demands on the user to translate their thinking into computer terms, is a better user interface.

The sample code on the disk includes a script to create and populate the qualifier tables we talked about in this article. The code and script was written for a Microsoft SQL Server database, so you may have to make adjustments to actually execute it on your system.

With this issue, I will be cutting back to quarterly instalments of *Surviving Client/Server*. Increasing work pressures make it more and more difficult for me to continue month in and month out to provide the quantity and quality of material I think you deserve. Look forward to another exciting episode in January!

```
Qualifiers                              QualifierFilters

quaRecID    quaID  quaFilterID quaCode   qlfDataField
1000        100    1           H         empSalaryOrHourly
1001        100    3           CON       empType
1002        100    3           REG       empType
1003        100    2           F         empFullOrPartTime
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
SELECT * FROM Employees
  WHERE
    (empSalaryOrHourly = "H") AND
    ((empType = "CON") OR (empType = "REG")) AND
    (empFullOrPartTime = "F")
```

Steve Troxell is a software engineer with Utimate Software Group in the USA. You can contact him at Steve_Troxell@USGroup.com