Surviving Client/Server: TTable vs TQuery

by Steve Troxell

One of the most frequently asked questions from Delphi programmers entering the world of client/server is "*Should I use* `TTable` *or* `TQuery` *to access client/server tables?*". It may be more productive to analyze each component's strengths and weaknesses so we may better understand when it is appropriate to use one or the other. In this issue we'll discuss the relative merits of these two components when they are used with client/server databases. Unless otherwise indicated, all discussion of 'tables' refers to a table within a client/server database.

TTable

The single biggest advantage of `TTable` is that it provides the most portable data access if you want the same front-end client to operate with multiple back-end servers. However, `TTable` really seems to have been designed with desktop databases like Paradox and dBase in mind. While it functions with SQL databases, it isn't the all-purpose table tool you might imagine.

Relational databases (ie SQL databases) are designed to be manipulated in sets. That is, operations are performed on one or more related records in one batch through a query, unlike more traditional databases which permit full freedom to navigate forwards and backwards and to move to specific records within the table. SQL tables are designed to be queried through the SQL language, which is what `TQuery` is good at. Unfortunately, the SQL language is not well suited for table navigation, which is what `TTable` is good at. In straight SQL syntax, there is no mechanism for a 'previous' record, or 'last' record, or other navigation concepts you take for granted in traditional databases. Some SQL servers provide 'scrollable cursors' which are navigable pointers within an SQL result set, but this is not widespread and not all vendors' cursors are fully navigable both backwards and forwards. Furthermore, these cursors typically operate upon a result set obtained through an SQL query, rather than being inherent to the table itself.

This paradigm shift in database design is a very important aspect to keep in mind when designing your front-end. Many database applications employ file browsers and let the user scroll freely through a particular file, but large client/server tables are not well suited to the browser concept. Some alternative approaches include populating a combo box with key values from the records, or providing a search dialog that lets the user narrow down the choices into a more manageable list. In either case, the user selects a particular record from the list and then the entire record can be retrieved and displayed.

The significant issue with `TTable` and client/server is that all `TTable` actions are ultimately translated into SQL queries to be processed by the server and you have lost control over how and when those SQL queries are made. For example, a frequent complaint among Delphi client/server users is that just opening a `TTable` can take upwards of 15 to 20 seconds. The truth is that the time it takes `TTable` to open an SQL table is directly proportional to the size of the table being opened. `TTable.Open` actually issues a series of SQL queries to obtain information about all the columns, indexes and defaults used in the table. This information is gathered only the first time the table is opened. After that, it issues a `SELECT` query that retrieves all of the columns and all of the rows from the table ordered by the index field (or the primary key if no index is specified). This is done each time the table is opened. Only the first row of this result set is returned to the client (or as many as are needed to populate data-aware controls), but the server is still going to build a result set in response to the query. So you can see that if you try to use `TTable` to read one value from one row of a very large table, just opening the `TTable` is going to take some time. This demonstrates the inherent weakness of `TTable` in client/server: you've lost control over the SQL being sent to the server to perform your tasks; you are at the mercy of the Borland Database Engine.

The upside is that `TTable` is generally fine for accessing small client/server tables of a few thousand records. You'll have to do some testing to determine where the point of diminishing returns is for your system. In some cases, such as the `TDBLookupCombo` component, you have no choice but to use a `TTable`. There are some third-party components similar to `TDBLookupCombo` that claim to accept `TQuery` datasets, but you have to look under the hood. Some of them simply copy the results of the query into a temporary table and use `TTable` to access it. In these cases you have to consider the overhead of creating, populating, and disposing of the temporary table.

Record Retrieval

Is there any advantage to using `TTable` to look up a record using `FindKey` rather than an SQL query? Not necessarily. First, `FindKey` is restricted to searching only indexed fields and the only way to search on more than one field is if the fields you are interested in are covered by one or more indexes. An SQL query, on the other hand,

can search based on any number of fields in the table, whether they are indexed or not. Granted, non-indexed fields are going to be slower to search on, but an SQL query performed at the server will be faster than a sequential search done through TTable in the client application.

Second, FindKey will ultimately send the server a SELECT statement with a WHERE clause to return the desired record; the same thing you would do with an SQL query. However, FindKey will fetch all of the fields of the record back from the server (despite how many fields you selected in the TTable's Fields Editor). With your own SQL query, you could request only the fields you were interested in. With large records where only a few fields are of interest, you could conceivably improve performance considerably.

FindNearest, however, is not easily emulated with SQL queries. Given a non-existent value to search for, FindNearest returns the record after the point in the sequence where the requested value should be. Assuming we are searching the Name field of the Customer table for the nearest match to 'Troxell, Steve', we can almost emulate this functionality with the query shown in Figure 1.

However, this won't give you just the one row after the point where 'Troxell, Steve' ought to be in the sequence; this gives you all the rows after 'Troxell, Steve'. It turns out that this is exactly the query used by TTable.FindNearest on SQL tables (the entire query is processed, but only the first row of the result set is sent back to the client).

A much more efficient way to emulate FindNearest would be to use a nested query to find the value and then retrieve the matching record; as shown in Figure 2.

In a nested query (sometimes referred to as a 'sub-query'), the inner SELECT statement is processed first and its results are fed into the outer statement. Here, the inner statement finds the lowest order value of the Name field that satisfies the condition that it is greater than or equal to 'Troxell, Steve'. This value is then used in the WHERE clause of the outer statement to retrieve that particular record. The server does much less processing to evaluate the aggregate function MIN in the inner statement than the brute force query which is used by TTable.FindNearest. Again, you don't have control with TTable.

Other problems with FindKey and FindNearest are the actions of Prior and Next after establishing a position with one of the Find methods. For example, FindKey is going to issue a query that returns a result set of at most (assuming no duplicate values) one record (the one that matches the key value). What happens if you use the Next method to scroll to the next record? There is no next record in the result set returned by FindKey, but there should be a next record in the actual table. In this case, TTable.Next issues a query very much like that shown above for FindNearest, with the key value you used in FindKey as the point of comparison. For example, if you were to use FindKey(['Troxell, Steve']) followed by Next, the query sent by Next is similar to Figure 1, but would be '>' instead of '>='.

TQuery

TQuery encapsulates one or more SQL statements, executes them, and provides methods by which you can manipulate the results. As you have seen from the past two issues, SQL provides some powerful capabilities in its own right. Queries can be divided into two categories: those that produce result sets (such as a SELECT statement) and those that don't (such as an UPDATE or INSERT statement). Use TQuery.Open to execute a query that produces a result set; use TQuery.ExecSQL to execute queries that do not produce result sets.

Once you call Open, the query is sent to the server to be processed and the first row of the result set is returned to the client application. Subsequent rows are generally not passed back to the client until you specifically request them by navigating through the result set. With a result set, you can do most of the same tasks you can with a TTable: you can navigate using the same First, Next, Prior, etc methods found in TTable or with the TDBNavigator component, you can link to data-aware controls via a TDataSource and you can edit the result set, provided the query conforms to certain restrictions (more on that in a minute).

Unidirectional Movement

An important little property to keep in mind when working with TQuery is the Unidirectional property. If your SQL server does not support bi-directional cursors (that is, you can only move forwards through the result set, not backwards), then you must set this property to False (the default) if you want your application to be able to navigate both forward and backward through the result set. For example, if you have a TDBGrid bound to the dataset, with Unidirectional set to False Delphi emulates bi-directional movement by buffering the records internally in the client application as they are returned from the server.

You may be concerned by this approach if you anticipate a large result set, because resources could be consumed rapidly. If you only need to traverse forward through the result set, then you can set Unidirectional to True, and Delphi will not buffer the records. If you only need to walk through a result set once, then doing so with Unidirectional set to True can be a bit faster since Delphi doesn't have

➤ *Figure 1*

```
SELECT * FROM customer
  WHERE name >= "Troxell, Steve"
  ORDER BY name
```

➤ *Figure 2*

```
SELECT * FROM customer
  WHERE name = (SELECT MIN(name) WHERE name >= "Troxell, Steve")
```

to go through the overhead of buffering the records. However, if you are going to traverse the result set many times, for example with a TDBGrid, setting Unidirectional to False may be a bit faster, since the rows will be read from the buffer instead of the server.

Some of the navigation methods are misleading when used on 'forwards only' datasets. For example, TQuery.Last actually moves beyond the end of the dataset and 'backs up' one record. Since this backwards movement is illegal, you will get a *'Capability not supported'* exception.

Modifiable Datasets

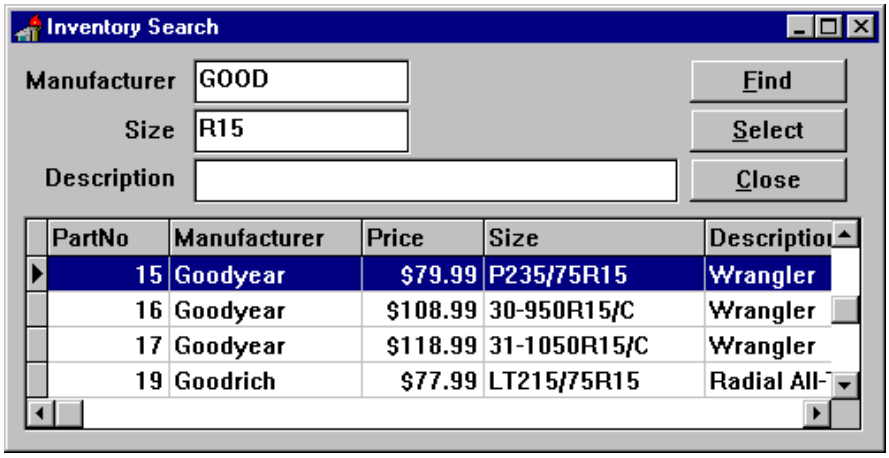
By default, TQuery returns a read-only result set and this is also a bit faster than a modifiable result set (which Delphi refers to as a 'live' result set). With a live result set you can use the same editing techniques (methods or data-aware controls) as with a TTable. To get a live result set, you must first set the RequestLive property to True. Second, you must use a SELECT statement that meets the following requirements:

- Conforms to Local SQL syntax (see the Delphi on-line help),
- Involves only one table or editable view,
- Does not use aggregate functions,
- Does not use an ORDER BY clause,
- The table must have a unique index if it is on a Sybase or Microsoft SQL server.

For live result sets, Delphi converts your SQL query into Query-By-Example syntax. The restrictions listed above ensure that your query can be converted into the QBE syntax. You can examine the CanModify property to determine if your TQuery has met all of the requirements to be editable.

Dynamic SQL Statements

The SQL query does not have to be hard-coded at design-time through the SQL property editor. The SQL property can be set through code at runtime to produce 'dynamic SQL'. This can be very powerful, since you can construct statements in response to user actions.



➤ *Figure 3*

```
procedure TForm1.FindBtnClick(Sender: TObject);
var  WhereClause: string;
begin
  WhereClause := '';
  if Mfg.Text <> '' then
    WhereClause := 'Manufacturer LIKE "' + Mfg.Text + '%"';
  if Size.Text <> '' then begin
    if WhereClause <> '' then WhereClause := WhereClause + ' and ';
    WhereClause := WhereClause + 'Size LIKE "%' + Size.Text + '%"';
  end;
  if Desc.Text <> '' then begin
    if WhereClause <> '' then WhereClause := WhereClause + ' and ';
    WhereClause := WhereClause + 'Description LIKE "%' + Desc.Text + '%"';
  end;
  if WhereClause = '' then
    raise Exception.Create('At least one search field must be entered');
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT * FROM Inventory WHERE ' +
    WhereClause + ' ORDER BY PartNo');
  Query1.Open;
end;
```

➤ *Listing 1*

```
SELECT * FROM Inventory
  WHERE Manufacturer LIKE "GOOD%" AND Size LIKE "%R15%"
  ORDER BY PartNo
```

➤ *Figure 4*

Take a look at the screen shot in Figure 3. This demonstrates a common technique used in client/server applications to allow users to pick from a list of choices after the list has been narrowed by one or more search criteria. This is an inventory search dialog for a tire store and the user is expected to supply one or more of: a part manufacturer, a part size, or a part description. The program constructs an SQL SELECT statement with a WHERE clause that reflects the values present in the edit boxes. If the user does not supply a value for one of the search fields, it isn't included in the WHERE clause.

When the user presses the Find button, the code shown in Listing 1 interprets the search values given and constructs an appropriate SQL query to find the matching records. The part manufacturer entry field will match against any record with a manufacturer that starts with the text the user entered, so we use the LIKE operator and append the SQL wildcard % to the text. The part size and part description fields can be any text anywhere in the field, so again we use LIKE and appropriate wildcard characters. With the screen filled out as shown in Figure 3, the SQL query shown in Figure 4 would be produced.

This is just a small example of the flexibility you have with dynamically created SQL statements. You can see how several different search fields can be added, and the user can fill out as many or as few as they desire to narrow their search. Another possibility would be adding radio buttons to control whether the search will be the intersection of matching records (by using a logical AND between operators) or the union of matching records (by using a logical OR between operators). It would be very difficult to construct a search engine with this much flexibility using just TTable methods.

Parameterized Queries

If you wanted a query to return a row that matches a specific key value, you could assemble the SQL statement at run-time as shown above, or you could hard-code the SQL statement through the property editor and use a substitution parameter for the key value:

```
SELECT * FROM atable
  WHERE cust_id = :customer_no
```

Here cust_id is the name of an integer field in the table atable. Customer_no is an arbitrary name for the parameter. It is called a substitution parameter because you will assign a value to the parameter and that value will be substituted in place of the parameter when the query is executed. For example, if you wanted to find the record with cust_no equal to 1234, you would assign the value to the parameter before executing the query (see Listing 2).

Notice that the parameter name is preceded by a colon in the SQL statement, but you do not use the colon when referencing the parameter with ParamByName.

You should be wary of using a lot of parameters on separate lines. For example, you may have an INSERT statement with a large number of fields and you may decide to list each field value on a separate line in the SQL property, as shown in Figure 5.

When a Delphi form containing a TQuery component is created at runtime, the SQL property is built one line at a time from the component stream. As the SQL is built, Delphi keeps an internal list of the parameters found. Unfortunately, as each new line of SQL code is added, the internal parameter list is destroyed and rebuilt from the first line of SQL down to the current line. To put this in perspective, if you have a TQuery component with an SQL statement having 25 parameters all on separate lines, when that TQuery is created at runtime, the internal list of parameters is destroyed and rebuilt 25 times with 325 total inserts into the list (only 25 will remain in the list after the last destroy/build iteration). This may not seem like much, but it does add a few seconds to the form creation.

In one of our projects, we had a small form created on demand at runtime. Adding three TQueries with a total of 75 parameters between them (one on each line) increased the form creation time from a fraction of a second to over 3 seconds. Not devastating, but quite annoying to the user. When we crammed all the parameters on just a few lines the time dropped back down to under a second.

'Preparing' Parameterized Queries

If you are going to re-use the same parameterized query repeatedly (for example in a loop), you can improve performance by explicitly 'preparing' the query once before the first use. All SQL queries ultimately get sent to the server where they are parsed, validated, compiled and executed. The TQuery.Prepare method sends the SQL statement to the server to complete the preliminary steps of parsing, validation, and compiling. In this way, these preliminary steps are performed only once instead of each time you execute the query from your Delphi application. If you use Prepare, you must also use Unprepare when you are through with the query. Prepare consumes resources in the database (and apparently in the application as well) and you must pair up each Prepare with an Unprepare to release those resources. Listing 3 illustrates this technique.

With a parameterized query, you can still change the values of the parameters in the compiled form of the query. However, if you change the SQL statement through code, you will have to 'prepare' it again. So, in Listing 3, if you chose to construct the SQL statement from code with the appropriate customer number already in place, you will defeat the advantages of preparing the query in advance. The Prepare method, if used, must be called after the SQL statement has been defined. If you define your SQL statements at design time with the property editor, you may want to put your Prepare call in the form's OnCreate event handler and the Unprepare call in the form's OnDestroy event handler.

➤ *Listing 2*

```
Query1.ParamByName('customer_no').AsInteger := 1234;
Query1.Open;
```

➤ *Figure 5*

```
INSERT INTO ATable (
  Name,
  Address,
  City,
  ( ...a whole bunch of fields... )
  DateOfBirth)
VALUES(
  :Name,
  :Address,
  :City,
  ( ...a whole bunch of parameters... )
  :DateOfBirth)
```

```
Query1.SQL.Clear;
{ the next line shows the SQL statement being used in the query }
Query1.SQL.Add('Select * from ATable where cust_no = :customer_no');
Query1.Prepare;
try
  for I := 1 to 1000 do begin
    Query1.ParamByName('customer_no').AsInteger := I;
    Query1.Open;
    try
      { do stuff with the returned record }
    finally
      Query1.Close;
    end;
  end;
finally
  Query1.Unprepare;
end;
```

➤ *Listing 3*

Conclusion

The prevailing wisdom for conventional databases (eg Paradox and dBase) is to use `TTable` wherever possible and only resort to `TQuery` when a task is beyond the capabilities of TTable. In the client/server world, however, most of the time the opposite is true.

Remember that paradigm shift in database methodology: client/server tables are meant to be manipulated in record sets. If you're building a conventional database application in anticipation of eventually upsizing to client/server, be very careful about your choice of data access. What may work just fine in a small local system may limp along like a three-legged dog in a high-volume client/server environment.

Contrary to much of the marketing hype, scaling up to client/server is not necessarily just a simple matter of changing the alias definition.

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on the internet at stevet@tpower.com and also on CompuServe at 74071,2207