The Data Shape OLE DB Provider

by Guy Smith-Ferrier

Much of the functionality of `TClientDataSet` is implemented in regular ADO recordsets, and, therefore, their ADOExpress `TDataSet` implementations. You can persist data using `SaveToFile` and `LoadFromFile`, create custom ('fabricated') recordsets and create briefcase applications. The Data Shape OLE DB Provider was introduced to ADO in v2.0 and it provides much of the remaining functionality of `TClientDataSet`.

In this article we will look at this OLE DB Provider and reveal what it is capable of and what additional functionality it has beyond `TClientDataSet`. We will look at hierarchical recordsets, parameterised hierarchies, reshaping, using aggregate functions, grouping data using `COMPUTE`, adding blank fields and creating custom shapes.

Hierarchical Recordsets

The most common use of the Data Shape OLE DB Provider (also known as MSDataShape) is to create hierarchical recordsets, also referred to as master/detail relationships, parent/child relationships and (by Microsoft only) the child side of the relationship is called a chapter. At first sight these hierarchical recordsets provide the same functionality as Delphi's own master/detail relationships and this is where we will start.

To use MSDataShape, add a `TADOQuery` to a form. In the Connection String editor select the MSDataShape OLE DB Provider. In `Data Source` on the `Connection` page enter an ODBC Data Source Name. MSDataShape can be used with any OLE DB Provider which returns rectangular data by setting the `Data Provider` value in the `All` page of the Connection String editor. For simplicity we will stick with ODBC and use the `Northwind` database which is installed with many Microsoft products. That's the `ConnectionString` done. Set `ADOQuery1.SQL` to

```
SELECT * FROM CUSTOMERS
```

open the query, and add a `TDataSource` and a `TDBGrid` to view the results. You shouldn't be too surprised at the result as it is exactly the same as executing this SQL normally. MSDataShape is looking for commands which are specific to itself. All other SQL statements are passed on to the OLE DB Provider which actually retrieves and updates the data. MSDataShape is looking for SQL commands which begin with `SHAPE`. Change the SQL to this:

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND ({SELECT * FROM ORDERS}
RELATE CustomerID TO
  CustomerID) AS ORDERS
```

then re-open the query and you should, once again, see the same result: that is, a list of all of the customers. However, if you cursor to the right hand side of the grid you will see a column full of `DATASET` values. Click it so that an ellipsis (. . .) appears. Click the ellipsis and a grid of all of the children belonging to the selected parent will be shown.

As Bob Swart has pointed out in the past, this may be clever but it isn't the best user interface implementation possible. The solution to this user interface obstacle using MSDataShape is the same as for `TClientDataSet`: add persistent fields. When you do this you will see an `ORDERS` persistent field at the bottom of the list. This is a `TDataSetField`. Now add a `TADODataSet` and set its `DataSetField` to `ADOQuery1ORDERS`. Show the new `TADODataSet` in a grid and you have a master/detail relationship working in same way as a regular Delphi master/detail relationship.

Take a look at the `SHAPE` command which we entered. It executes two `SELECT` statements immediately the dataset is opened:

```
SELECT * FROM CUSTOMERS
```

and

```
SELECT * FROM ORDERS
```

The MSDataShape provider then forms a relationship between the two result sets based on the criteria supplied in the `RELATE` clause.

The first benefit of MSDataShape is that it is efficient. Often programmers will use an SQL `JOIN` to achieve a similar result to that of MSDataShape using something like the following command:

```
SELECT * FROM Customers, Orders
WHERE Customers.CustomerID=
  Orders.CustomerID
```

In the `Northwind` database the `Customers` table holds 91 records of 268 bytes each. The `Orders` table holds 830 records of 206 bytes each. This `SELECT` command retrieves 393,420 bytes (that is, 830*(268+206)). The equivalent MSDataShape command retrieves under half of this amount, 195,368 bytes (that is, (91*268)+(830*206)). Also consider that MSDataShape is simply using the server to retrieve data. It is not using the server to perform any other work. The additional work of forming the relationship between the parent and child is performed by the client. Consequently, this solution is more scalable because the load has been moved from the server to the client.

Of course, this doesn't take into account orphaned children. An orphaned child is one which has no parent. With modern databases having built in referential integrity this is unlikely to happen as frequently as it once used to, but it could occur by design. Assume that your `Customers` table holds a field called `County` which is a lookup into a `Counties` table. The

counties of England are a relatively volatile affair with counties ceasing to exist (eg Rutland and Avon) and new counties arriving every few years (or like Rutland being resurrected). Consequently, it might be acceptable to allow the value in the `County` field to be blank. An SQL `JOIN` between `Counties` and `Customers` would retrieve only those customers that had a `County` but a `SHAPE` would return all of the customers regardless and would include data which was inaccessible through the hierarchy.

Unfortunately the most recent ADOExpress patch (October 2000) at the time of writing added a problem which previous patches didn't suffer from. If you set the `LockType` to batch optimistic and use MSDataShape your result set will appear to be empty.

Parameterised Hierarchies

The same net result as above can be achieved with the following `SELECT` statement:

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND
({SELECT * FROM ORDERS WHERE
  CustomerID=?}
RELATE CustomerID TO PARAMETER
  0) AS ORDERS
```

This is a parameterised hierarchy. It uses the `CustomerID` in the parent as a parameter to be passed to the child statement (parameters in SQL are marked with a `?`). This differs from the previous example, not in the data which is returned, but in the *way* the data is returned.

When the dataset is opened the `SELECT * FROM CUSTOMERS` command is executed but the `SELECT * FROM ORDERS...` is not. Instead, the child statement is executed each and every time a new parent is selected.

There are a number of consequences to this approach. Parameterised hierarchies are faster to open than regular hierarchies because they retrieve less data initially. In addition, they do not suffer from retrieving orphaned children. The data in the child dataset is more fresh because it is retrieved when it is needed. However, parameterised hierarchies cannot be disconnected from the database, cannot be persisted (that is, saved locally), cannot be passed across process boundaries and cannot be reshaped (we will cover reshaping shortly).

The ADO Recordset interface, around which the ADOExpress `TDataSets` are based, supports a dynamic property called `Cache Child Rows`. This property allows a regular hierarchical recordset to retrieve its children on an as needed basis in the same way as a parameterised hierarchy. By default it is `True`, so all records are retrieved when the parent is opened. Unfortunately, this dynamic property won't help you because you need to set it on an existing recordset before it is opened. ADOExpress doesn't provide you with a means to access the recordset after it has been created but before it is opened.

Back to the parameterised hierarchies. At first sight it seems that the disadvantages outweigh the advantages, but consider the following regular `SHAPE` hierarchy:

```
SHAPE {SELECT * FROM CUSTOMERS
   WHERE Country='UK'}
APPEND ({SELECT * FROM ORDERS}
RELATE CustomerID TO
  CustomerID) AS ORDERS
```

This applies a `WHERE` clause to `CUSTOMERS` so that only UK customers are retrieved. Unfortunately, the child `SELECT` statement retrieves all orders including those for customers outside the UK. Clearly this is wasteful. One way to solve this is to use a parameterised hierarchy:

```
SHAPE {SELECT * FROM CUSTOMERS
  WHERE Country='UK'}
APPEND ({SELECT * FROM ORDERS
  WHERE CustomerID=?}
RELATE CustomerID TO PARAMETER
  0) AS ORDERS
```

A better way to solve this is to avoid using a parameterised hierarchy (because of the significant restrictions it imposes) and apply the `WHERE` clause to each child `SELECT` statement as in Listing 1.

Parent, Child And GrandChild Hierarchies

You are not restricted to just one level of parent/child relationships. You can build hierarchies with many siblings (parent/child/child) and hierarchies with many levels of child (parent/child/grandchild). The syntax is no different from a simple parent/child relationship but, like most complicated SQL statements, it does take a little while to formulate and even longer to read and understand.

Assume that we have another hierarchy represented by the following `SHAPE` command:

```
SHAPE {SELECT * FROM ORDERS}
APPEND ({SELECT * FROM
  [Order Details]}
RELATE OrderID TO OrderID) AS
  OrderDetails
```

(The square brackets around `Order Details` cope with the embedded space in the table name. A better solution to this problem is simply not to embed spaces in your table names or field names.)

The resulting `SHAPE` command for `Customers`, `Orders` and `Order Details` is shown in Listing 2.

The trick is to formulate each statement separately and to use round brackets around the inner `SHAPE` commands.

```
SHAPE {SELECT * FROM CUSTOMERS WHERE Country='UK'}
APPEND ({SELECT ORDERS.* FROM ORDERS, CUSTOMERS
WHERE Orders.CustomerID=Customers.CustomerID
AND Customers.Country='UK'}
RELATE CustomerID TO CustomerID) AS ORDERS
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND (
(SHAPE {SELECT * FROM ORDERS} AS rsOrders
APPEND ({SELECT * FROM [Order Details]}
RELATE OrderID TO OrderID) AS OrderDetails)
RELATE CustomerID TO CustomerID) AS ORDERS
```

Reshaping

Reshaping is one of the few enhancements which were made to MSDataShape in ADO 2.1 (there were no further enhancements in ADO versions 2.5 or 2.6). Reshaping allows you to use elements of an existing SHAPE command for other purposes. In order to reuse elements they must be named. Change the original SHAPE statement so that the child SELECT statement is given the name rsOrders:

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND ({SELECT * FROM ORDERS}
  AS rsOrders
RELATE CustomerID TO
  CustomerID) AS ORDERS
```

Now the rsOrders recordset can be used in its own right. The visibility of the name is connection-wide so the same connection must be shared between the two datasets. Add a TADOConnection to take the place of the ConnectionString in ADOQuery1.

Add a TADOQuery for the same connection and set its SQL property to:

```
SHAPE rsOrders
```

Now the Orders recordset can be used independently of the hierarchy.

Aggregate Functions

One of the great features of TClientDataSet is maintained aggregates. If you've never used maintained aggregates before then you should take a look at them.

➤ *Table 1*

Function	Description
Min	Minimum value
Max	Maximum value
Count	Number of rows
Sum	Sum of rows
Avg	Average of rows
STDev	Standard Deviation of rows
Any	Any value of a column (assuming all values are the same)

```
SHAPE {SELECT * FROM ORDERS}
APPEND ({SELECT [Order Details].*, UnitPrice * Quantity AS ItemTotalValue
  FROM [Order Details]}
RELATE OrderID TO OrderID) AS OrderDetails
```

➤ *Listing 3*

Have you ever been asked to total a column in a grid ? If so then TClientDataSet's maintained aggregates are a good way to solve this problem. MSDataShape has a similar set of features and here I'll explain how they work.

Assume that we have the regular SHAPE command in Listing 3. The only special detail here is that the child recordset contains a calculated column, ItemTotalValue, which is UnitPrice * Quantity. Now add the following clause to the end of the command:

```
, SUM(
  OrderDetails.ItemTotalValue)
  AS OrderTotalValue
```

The SUM function sums a set of values in a child recordset. The expression in the SUM function must refer to a value in a named recordset in the SHAPE command. Recordsets do not have default names so you must explicitly assign them one using AS.

Now add persistent fields to the dataset and add a grid to show the orders. Add another TADODataSet and grid to show the details. Run the program and make changes to the UnitPrice of any of the Order Details. As you start making the change you won't be at all surprised to see the gutter on the left hand side of the grid change to an I-beam to indicate that the record is in edit mode. But look at the parent record in the grid above it and you will see that the parent has also gone into edit mode. Now complete the change to the UnitPrice and you will see that the ItemTotalValue field in the child is not updated to reflect the new change. This shouldn't be too surprising because this field was calculated by the DBMS and not by ADO. However, you can update ItemTotalValue manually, so go ahead and do this. Move off the record to post it and the parent record's OrderTotalValue is not updated. But if you go back to the parent record and move away from the parent record to post it the OrderTotalValue column will be correctly updated.

The full list of aggregate functions supported by MSDataShape is shown in Table 1.

Now add the following clause to the end of the SHAPE command:

```
, COUNT(OrderDetails.OrderID)
  AS OrderCount
```

Clearly this is just a simple count of the number of items which the order has. Run the program and add a new Order Detail. You will see the same behaviour as before where the parent record also goes into edit mode when the child record is added and, when the parent record is posted, the count changes to reflect the new number of items.

Unfortunately, all is not well here. Try deleting an Order Detail. The parent record does not go into edit mode and it does not register that the child has been deleted, and so both the SUM and the COUNT are wrong. They do catch up correctly the next time they are updated, as a result of a child record being added or edited, but that isn't much consolation. Sadly, this is a bug in ADOExpress and not in ADO. It can be rectified by adding a BeforeDelete event to the child to ensure that the parent is updated:

```
procedure
  TForm1.ADODataSet1BeforeDelete(
  DataSet: TDataSet);
begin
  ADOQuery1.Edit;
  ADOQuery1.Post;
end;
```

Grouping Data Using COMPUTE

There is another interesting keyword in the SHAPE language and it is COMPUTE. It is very similar to SQL's GROUP BY clause. It is used like this:

```
SHAPE {SELECT * FROM Employees}
  AS Employees
COMPUTE Employees BY Title
```

If you view the result of this SHAPE command in a grid it will be almost identical to the following SQL command:

```
SELECT TITLE FROM EMPLOYEES
GROUP BY TITLE
```

This SQL command gives you a list of all of the unique titles in the Employees table. This can be a very useful device because databases often do not include all of the lookup tables which they should. Such is the case with the Northwind database: there is no table for Titles. The GROUP BY clause allows, amongst other things, a virtual table to be created from the existing data.

The COMPUTE clause does all that the GROUP BY clause does but with an important difference: it includes the records which form part of each group. If you add persistent fields to the dataset for the SHAPE command you will see an Employees TDataSetField. Just as with the hierarchical recordsets, add a TADODataSet and set its DataSetField to the Employees persistent field and you will be able to view the records which make up the group.

Adding Blank Fields

Have you ever wished that calculated fields were not read-only? From time to time a problem comes along where it would be really useful if you could add a couple of extra temporary fields to a dataset. Calculated fields don't cut it. You can set their values in the TDataSet.OnCalcField event but, apart from that, they are read-only. MSDataShape provides you with a means to add temporary fields to a result set which are visible only from the result set. The following SHAPE command does just that:

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND NEW adBoolean AS
  Selected
```

It adds a new Boolean field called Selected to the CUSTOMERS result set. This field can be read and written to in the same way as any other field but the underlying table is never updated. When the result set is closed the values entered into the temporary fields are permanently lost. The Selected field in this SHAPE can be used to allow the user to mark a number of records for some kind of action. You can, of course, already achieve this using TDBGrid by including dgMultiSelect in the Options but the selection is a little bit too transient for my liking because your selection is cleared the moment you stop selecting.

Custom Shapes

A logical extension of the ability to add additional blank fields to existing recordsets is to create a recordset which consists entirely of blank fields and does not attach to any such data source. The following SHAPE command achieves this:

```
SHAPE APPEND
NEW adVarChar(2)  AS STATE,
NEW adVarChar(30) AS NAME
```

As there is no original source of data you must set the ConnectionString's Data Provider (in the All page) to None.

There is a very good reason why you might want to create such an empty recordset. Consider all of those temporary tables which us programmers love to create. Many programmers want to copy data to a local store and work on the data locally for various reasons. At this point it usually gets a bit messy, with everyone creating temporary tables and worrying about where to put them safely and how to tidy up the temporary tables when the application crashes.

Custom Shapes are a better solution because the temporary table is completely held in memory. There is no worrying about creating a local table and having a local DBMS, or worrying where the temporary table can be safely located and how to tidy it up. Of course, the whole table is held in memory and there is no denying that this places an upper limit on the size of the table.

You can achieve this same affect using TClientDataSet and TADODataSet by defining their FieldDefs at design-time and then right clicking and selecting Create DataSet.

These two solutions provide almost exactly the same benefits as Custom Shapes. However, these datasets cannot be opened: they must be created. The distinction is not important to all applications but if you ever need to set Active to False and then set Active to True again with a fabricated TClientDataSet or TADODataSet then you will get an error. Unfortunately, the Delphi 5 IDE does exactly this when you create persistent fields at design-time (note that this is not the case with versions prior to Delphi 5). Using a Custom Shape you do not suffer from these limitations as they can be opened and closed just like any other dataset. One of the consequences of this difference is that Custom Shapes are more suitable for use as in-memory lookup tables than TClientDataSet or TADODataSet.

Conclusion

MSDataShape offers a collection of features which would take quite some additional effort to recreate without its aid.

It can be used with any OLE DB Provider and so it is an excellent example of a consumer. The use of hierarchical recordsets can improve performance. Reshaping allows you to look at data from different viewpoints. Aggregate functions provide much of the same functionality as TClientDataSet's aggregate functions. In

addition, `COMPUTE` provides the same facilities as `GROUP BY`, but it lets you get at the records in the group. You can also add read/write temporary fields to existing recordsets and you can create custom recordsets to replace temporary tables.

All in all that's a lot of features for just one OLE DB Provider, so do enjoy exploring the possibilities in your own applications!

Guy Smith-Ferrier is a Senior Delphi Consultant for Borland's Professional Services Organisation in the UK. He continues his ambition to play the piano better than a deaf one-armed monkey but is beginning to realise that he has met his match. You can contact Guy at gsmithferrier@capellasoft.com