**Apress™**
**Books for Professionals by Professionals™**

**Sample Chapter: "Recordset Recursion and Data Shaping"**
**(pre-production "galley" stage)**

# Serious ADO: Universal Data Access with Visual Basic

by Rob Macdonald
ISBN # 1-893115-19-4

CHAPTER 8

# Recordset Recursion and Data Shaping



**AS ADO HAS MATURED,** it has expanded its range to include models of data that it originally could not represent. Early versions of ADO presented all data in the form of tabular Recordsets. While it was possible for Providers to stretch the tabular model by placing compound data structures such as arrays in a single Recordset cell, ADO Recordsets were pretty much two-dimensional.

The first extension of ADO beyond this model was the introduction of Data Shaping. Its ability to embed Recordsets within Recordsets, popularly known as *hierarchical Recordsets*, is the subject of this chapter. All Data Shaping takes place using the MSDataShape Provider that is a standard part of an ADO installation.

A degree of mystique has developed around Data Shaping, mostly generated by authors who find any syntax that involves more than placing a dot between an object and a property or method name confusing or counterintuitive. We have come to realize that OLE DB Providers can expose command languages of their own making, and the MSDataShape Provider is just one of a growing number of Providers to do so. You'll see that the syntax of the SHAPE language is far, far simpler than the syntax of SQL, and that while the combinations it supports can result in rather convoluted statements, its very small number of keywords and syntax forms make it easy to learn.

In Chapter 13 you'll also see that VB comes with a built-in wizard for creating MSDataShape commands, so there is little excuse for not embracing this technology, especially as a number of user interface components, including the VB6 Report Designer and the Hierarchical FlexGrid control, have been created to exploit it.

The major challenge presented by Data Shaping is to learn when it's applicable and what it gives us that is new. The answer to both of these questions arises from the main purpose of Data Shaping, which is to take two or more logically tabular structures and combine them into a single Recordset that maintains the structural relationship between the original sets of data.

If this sounds a little too abstract, think about what happens when you use SQL to execute a standard join. An SQL join takes two or more logically tabular structures (for example, records from two tables), and as a result of combining them, flattens them into a single tabular structure. This is what joins are meant to do, and often it's exactly what we want. In contrast, Data Shaping doesn't flatten the data sets it combines. Instead, it maintains their hierarchical relationship (assuming they have one). When this is what we want, it's time to use Data Shaping.

## The Idea of Data Shaping

Data Shaping allows you to create two types of Recordset hierarchies:

- *Relation-based hierarchy*: Two Recordsets that share a common key are formed into a parent-child hierarchy indexed on that key.

- *Group-based hierarchy*: One Recordset becomes a child of its own aggregated data. In other words, you can view its totals, averages, and other statistics at one level, and drill down to see the base data when required.

These two basic types of hierarchies can be combined with each other or with themselves to create sophisticated drill-down structures. Relation-based hierarchies can be parameterized, which causes data lower down in the hierarchy to be retrieved on an "as needed" basis. This is particularly useful when creating deeply nested hierarchies of the type that might be used in a management information system.

All Recordsets created using the MSDataShape Provider have client-side cursors. Hierarchical Recordsets can be updateable, with updates supported at any level in the hierarchy. While this sounds a bit magical, once you have seen how a hierarchical Recordset is constructed, you'll see exactly how updating works.

One easy way to visualize how Data Shaping works is to simulate a hierarchical Recordset using regular ADO techniques. Consider the following two Recordsets:

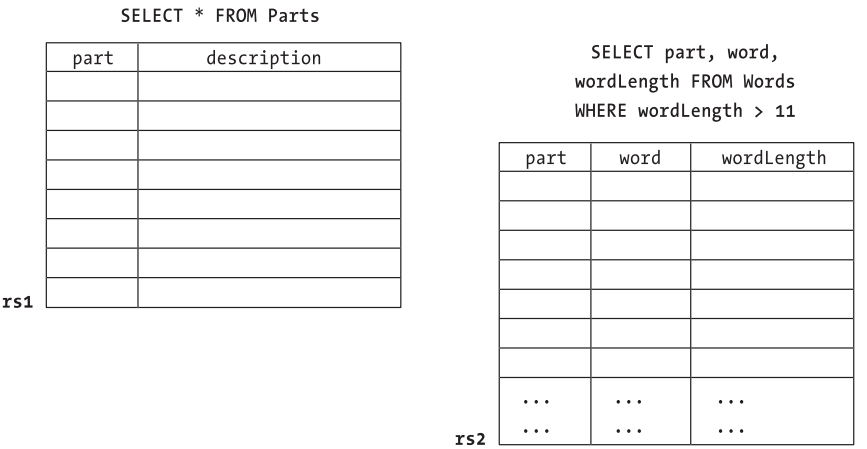



*Figure 8-1. Simulating Data Shaping using regular Recordsets*

Now consider using the rs1_MoveComplete event to set a filter on rs2 so that the only records visible in rs2 are those that have the same "part" field as the currently selected record in rs1. Here's the code that would achieve this:

```
Dim cn As New Connection
Dim WithEvents rs1 As Recordset
Dim rs2 As Recordset

Private Sub cmdFilter_Click()

cn.CursorLocation = adUseClient      <-- use client-side cursors
cn.Open "File Name=c:\MuchADO.udl"
Set rs1 = cn.Execute("SELECT * from Parts")
Set rs2 = cn.Execute("SELECT part, word, wordLength " & _
                     "FROM Words WHERE  wordLength > 11")
rs2("part").Properties("OPTIMIZE") = True
                                     <-- build an index on the
                                         part field of rs2
End Sub
```

```
Private Sub rs1_MoveComplete( _
        ByVal adReason As ADODB.EventReasonEnum, _
        ByVal pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        ByVal pRecordset As ADODB.Recordset)
rs2.Filter = "part = '" & rs1!part & "'"
End Sub
```

filter rs2 according to the current record in rs1

This is a pretty good simulation of a simple relation-based hierarchy. You could bind these two Recordsets to two Data Grid Controls using two ADO Data Controls, and create a display such as this:[1]

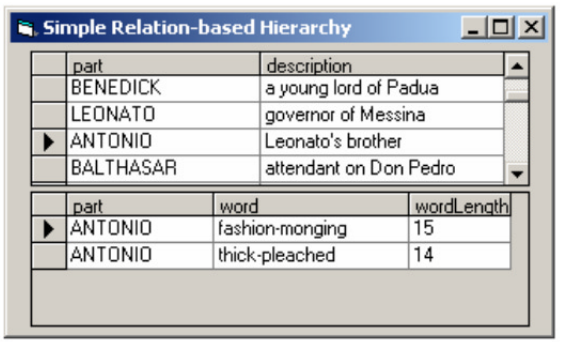



*Figure 8-2: Form displaying simulated Data Shaping*

Clicking a row in the upper grid (rs1) causes the lower grid (rs2) to display the big words of the selected part. For the sake of completeness, here's the binding code for four appropriately named controls:

```
Set dgRS1.DataSource = adcRS1
Set dgRS2.DataSource = adcRS2
Set adcRS1.Recordset = rs1
Set adcRS2.Recordset = rs2
```

bind the Data Grids to the Data Controls

bind the Recordsets to the Data Controls

Now that you have seen how to simulate Data Shaping, it's time to look at the real thing, and make use of the MSDataShape Provider. When you use Data Shaping to create a hierarchical Recordset based on the two previous queries, the resulting structure looks like this:

1. Which shows that Antonio can only construct long words by applying some blatantly thick-pleached fashion-monging.

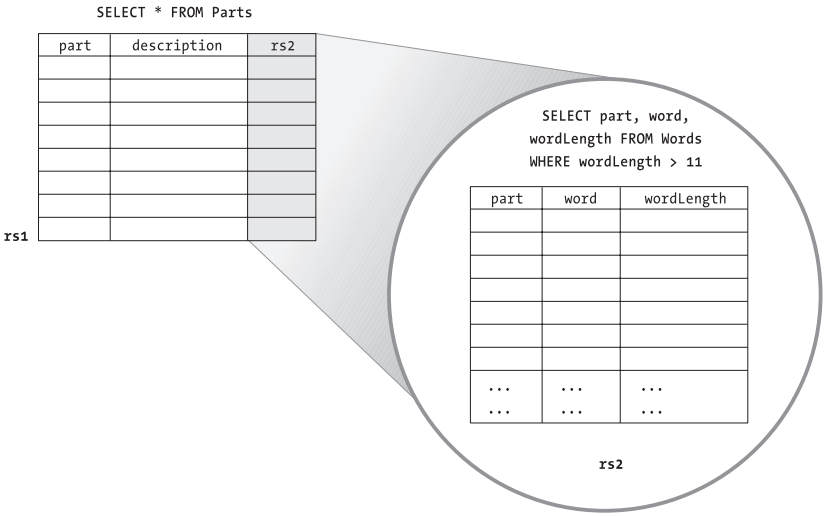



*Figure 8-3. Structures created by a Data Shaping command linking two simple Recordsets*

What is happening here is that MSDataShape creates the two Recordsets based on a single command that contains both SQL statements, along with a relationship statement that links the part fields in the two Recordsets. MSDataShape automatically indexes the part field in the child Recordset (rs2), and then appends a new field onto the parent Recordset (rs1) with a name of your choice (which in this case is "rs2").

This appended rs2 field has a data type of adChapter. A *chapter* is a subset of records from the child Recordset that relate to a single record in the parent Recordset. If you printed the chapters for each record in the parent Recordset, you would print the entire child Recordset, or at least all those records in the child for which a related record exists in the parent.

However, the child Recordset always has a filter applied to it, based on the value of the part column in the parent Recordset. This filter makes it look as though there is a different child Recordset associated with each record in the parent Recordset, when in reality, there is only one child Recordset. Regardless of the record you are positioned on in the parent, the Chapter-type Field always exposes the corresponding records in the child according to the defined relationship. The filter is internal to the hierarchical Recordset, and so it can't be modified using rs.Filter (which leaves rs.Filter free to be used to create subsets of the currently visible child records).

Assuming the resulting hierarchical Recordset has been assigned to rs1, the following code would access the (filtered) child Recordset for the current parent record:

```
Dim rs2 as Recordset
Set rs2 = rs1("rs2").Value
```

Now `rs2` can be used like a regular Recordset (because it is one). Note that it's important to use `.Value` explicitly, because otherwise, VB would attempt to assign a Field object to a Recordset variable, which would result in a type mismatch error.

To manipulate all the data in this hierarchical Recordset, you need to know about its hierarchy so that you can use the Chapter-type Field to drill down to the detail. The hierarchical Recordset maintains the natural relationship between the two Recordsets, whereas a standard SQL join flattens the resulting data, destroying the natural relationship. In cases where it makes sense to maintain this relationship, the Data Shaping approach is often a better way to manage the data than using an SQL join. The scenario just discussed provides a good example. It's a master-detail relationship, and making the relationship explicit is a positive advantage.[2]

You could retrieve the same data using a standard SQL join, as follows:

```
SELECT P.part, P.description, W.word, W.wordLength
FROM Parts P,Words W
WHERE  W.wordLength > 11
AND P.part = W.part
```

The natural structure of the resulting Recordset is the standard, flat, tabular form. It isn't very easy to present this representation of the data in a master-detail style of user interface.

However, consider which approach is the most convenient if the user wants to list all the big words from the play and the parts that are responsible for speaking them. Given this requirement, the flat structure represented by the join is more appropriate. The user may want to be able to sort all the big words alphabetically, for example. This won't be possible using the hierarchical Recordset because the child Recordset (which contains all the Words) is always filtered by the parent (which is based on Parts). Of course, you could always reverse the relationship and make the Parts query into a child of the Words query. This means that you would have to drill down just to see the single part description for the current word, which is an unnecessary complexity. For this type of usage, the hierarchy just gets in the way.

Data Shaping is not a replacement for joins. It's an alternative to be used when it makes sense to preserve and exploit the natural relationship between two sets of data. It so happens that this is often exactly what you want to do. You'll also see that it can do things that are very awkward to do using standard SQL.[3]

---

2. In Chapter 13 you'll see how both the VB6 Data Environment and the Microsoft Hierarchical FlexGrid make the task of creating user interfaces that reflect this hierarchical relationship almost trivial.

3. At the same time, there are things you can do in SQL that can't be done by Data Shaping. One example is an outer join.

One final point worth noting from the previous comparison of a joined and a hierarchical Recordset is that in the joined Recordset, the description column appears in every record, whereas in the hierarchical Recordset, it appears only once for each part rather than once for each word. In some cases, the hierarchical approach can be more efficient because it involves less duplication of data, although this fact is balanced by the need to issue two SQL statements to the database instead of one.

The beauty of the hierarchical approach becomes clear when you start creating more complex structures of Recordsets. For example, a single command (and therefore a single Recordset variable) can provide access to the following type of structure:

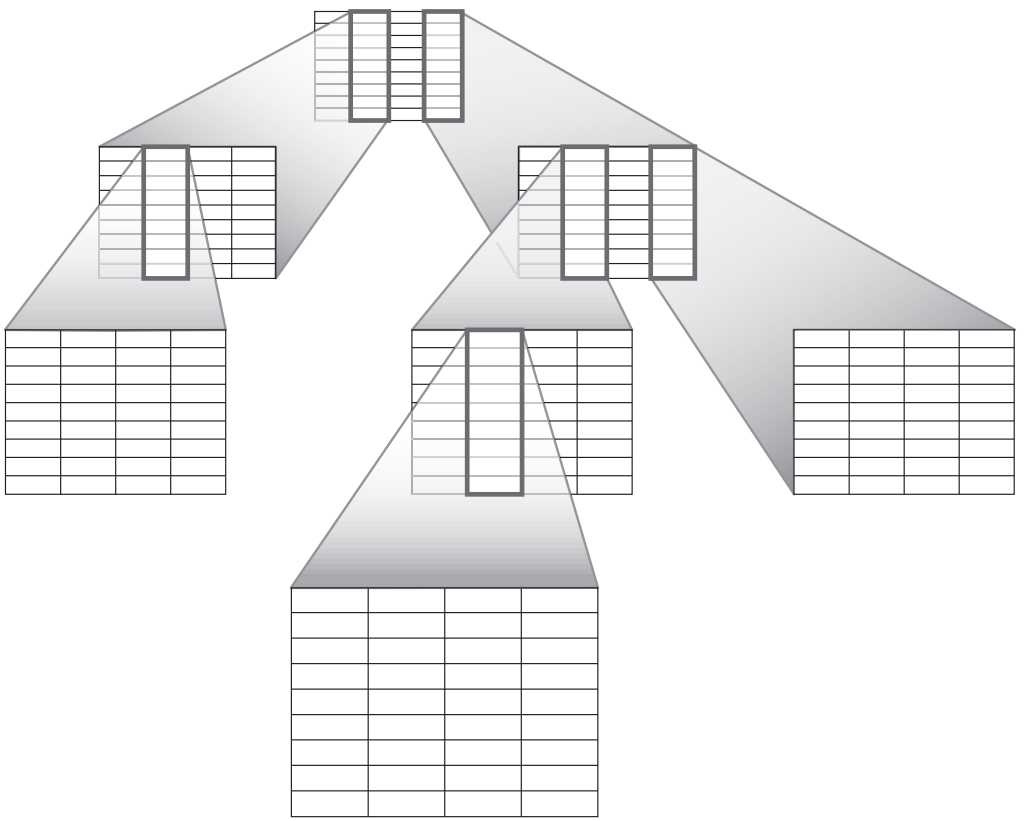


*Figure 8-4. A single hierarchical Recordset can represent a complex data structure.*

By passing an appropriate command to MSDataShape, it will execute the seven embedded commands required to construct this structure, build and index seven client-side Recordsets, and append six Chapter-type Fields to four Recordsets. The entire data structure can be held in memory for

highly responsive drill-down and disconnected or persisted efficiently as a single structure.

Alternatively, a parameterized approach can be taken, in which case MSDataShape only builds the parent Recordset and the children are constructed as needed on a chapter-by-chapter basis. This type of hierarchy can't be disconnected (for obvious reasons), but it requires less memory and takes less time to create initially.

Either way, changes made by users at any level in the hierarchy are automatically saved back to the data source, using standard client-side modification techniques on the appropriate Recordset.

You have seen how you can *almost* simulate relation-based hierarchies by using filters. However, the standard model provided by Data Shaping is more convenient, more amenable to standard processing and data binding techniques, and provides the added power of being able to parameterize child Recordsets to provide "just-in-time" data retrieval, which is highly appropriate for larger, connected hierarchical Recordsets. We'll look at group-based hierarchies later in the chapter.

### In This Section

We examined how hierarchical Recordsets are constructed, and we compared the hierarchical approach to a more traditional type of query such as an SQL join.

## Creating the Connection String

We discussed the *idea* of hierarchical Recordsets in the last section. Let's now go about creating them.

The first challenge is to create the required Connection object. The Provider name for Data Shaping is always MSDataShape. The issue here is that MSDataShape doesn't have any data of its own—it merely provides a service based on Recordsets retrieved from somewhere else. Therefore, as part of the connection process, we have to tell MSDataShape about the Provider that will be used to source the data.

There are several ways of doing this, and fortunately, all of them are simple. Here's some code for one such option:

```
Dim cn As New Connection
cn.Provider = "MSDataShape"
cn.Properties("Data Provider") = "SQLOLEDB"
cn.Properties("Data Source") = "POLECAT"
cn.Properties("Initial Catalog") = "MuchADO"
cn.Open , "sa", ""
```

use the Data Provider property to set the underlying Provider name

Any reasonable combination of dynamic properties, connection string key-value pairs, and cn.Open arguments can be used. The key points are to make the Provider MSDataShape, use the Data Provider dynamic property to set the underlying data source, and set all other required Data Provider properties as though they were properties of MSDataShape. MSDataShape will then forward these on to the Data Provider.

You can also use the Data Link Properties window to create a .udl file. The following steps explain how:

1. Select MSDataShape in the Provider tab.

2. Select the All tab. Double-click the Data Provider property and set the appropriate Data Provider name (or leave MSDASQL as the default, if appropriate).

3. Either fill in the remaining properties as you normally would if the Data Provider were the Provider, or fill in the remaining properties via the All tab.

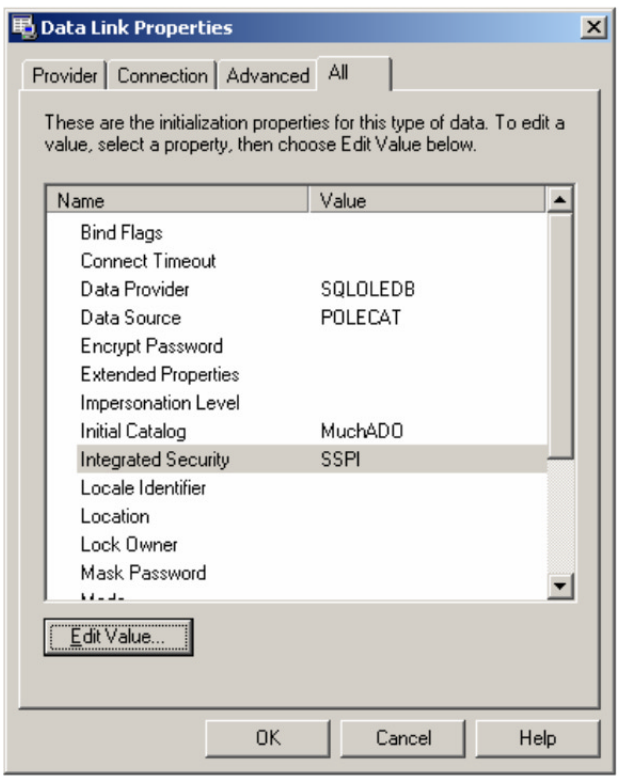



*Figure 8-5. Configuring a .udl file for the MSDataShape*

As soon as you open a Connection to MSDataShape, a connection to the Data Provider is created, and it's kept open until the MSDataShape connection is closed or released. You can therefore apply the same connection management and pooling rules to an MSDataShape connection as you would to the underlying Data Provider.

### In This Section

We saw how to create a connection to MSDataShape and configure the Data Provider.

## Relation-based Hierarchies

Now that we have a connection, we can start looking at the SHAPE language.

The basic form of a SHAPE command for creating a relation-based hierarchy is this:

```
SHAPE {<parent-command>}
    APPEND ({<child-command>}
        RELATE field TO field
        )
```

MSDataShape doesn't care about the command substrings within the curly brackets. It simply lifts them out of the command and fires them separately at the Data Provider. The `APPEND` statement has two parts, both contained within parentheses. The first is the `child command`. The second is a `RELATE` statement, which identifies the fields in the parent Recordset and the child Recordset that are linked. The field names don't need to be the same, but the data types must correspond. There can be multiple pairs of related fields for a "compound key."

As you have seen, MSDataShape will use the child field to index the child Recordset, and it will use the parent field to build the internal filter on the child Recordset, based on comparing the parent field to the child field. It will also append a Chapter-type column to the parent Recordset with the default name "Chapter1".

This is a very simple syntax, wrapped around some command strings embedded in curly brackets. It's common to give names or aliases to each Recordset, which makes the overall command string clearer to follow, if more verbose. Here's the general form with aliases:

```
SHAPE {<parent-command>} as ParentName
    APPEND ({<child-command>} as ChildName
           RELATE field TO field
           ) As ChapterFieldName
```

With this form, it becomes possible to identify the individual Recordsets by name, and you'll see how these names can be used shortly. By placing a `ChapterFieldName` after the `APPEND` statement, you can override the default name allocated to the Chapter-type Field with your own name. If you don't provide a `ChapterFieldName`, but do provide a `ChildName` , this `ChildName` will be used as the `ChapterFieldName`.

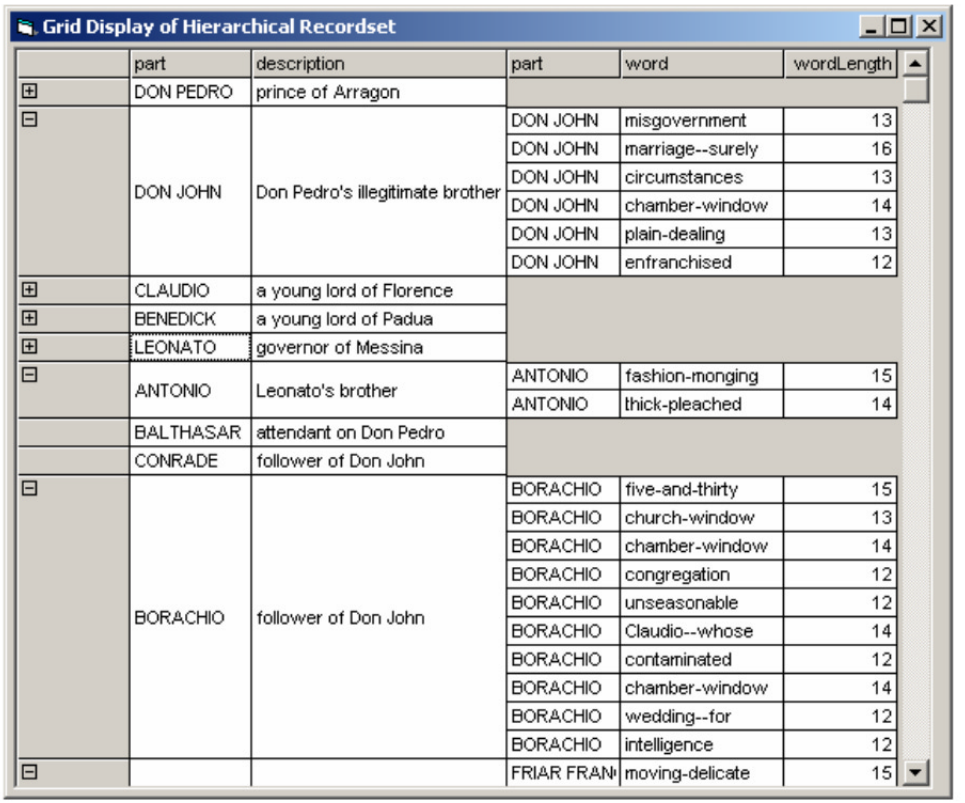
Here then, is some code using an actual SHAPE command, using the same SQL queries that were used in the Data Shaping simulation presented earlier in the chapter:

```
Dim sCommand As String
Dim rs As New Recordset
Dim cn As New Connection

cn.Open "File Name=c:\shape.udl"
sCommand = "SHAPE {SELECT * FROM Parts}" & _
               "APPEND ({SELECT part, word, wordLength " & _
                         "FROM Words WHERE  wordLength > 11 }" & _
               "RELATE part TO part ) As Words"

rs.Open sCommand, cn
```

This code creates a hierarchical Recordset, but it doesn't do anything with it. We could assign it to the Recordset property of a Microsoft Hierarchical FlexGrid (HFlexGrid) control (covered in Chapter 12), in which case we might end up with a display that looks like Figure 8-6.

There you can see the two Fields from the parent Recordset, and the three Fields from the child Recordset. The key to understanding this screen is to look at the left-most grid column, containing the + and – boxes. A + box is used to expand the parent record to show the child records (and therefore indicates that the child records are not currently being shown). A – box can be used to collapse a currently expanded parent to remove the child records from the display. A parent record that doesn't have a box has no child records.

We'll look at two ways of processing this Recordset in code. The first technique will be specific to this particular Recordset. The second technique is a generic version of the first technique, and can be used to navigate the structure of any hierarchical Recordset.




| | part | description | part | word | wordLength |
|---|---|---|---|---|---|
| ⊞ | DON PEDRO | prince of Arragon | | | |
| ⊟ | DON JOHN | Don Pedro's illegitimate brother | DON JOHN | misgovernment | 13 |
| | | | DON JOHN | marriage--surely | 16 |
| | | | DON JOHN | circumstances | 13 |
| | | | DON JOHN | chamber-window | 14 |
| | | | DON JOHN | plain-dealing | 13 |
| | | | DON JOHN | enfranchised | 12 |
| ⊞ | CLAUDIO | a young lord of Florence | | | |
| ⊞ | BENEDICK | a young lord of Padua | | | |
| ⊞ | LEONATO | governor of Messina | | | |
| ⊟ | ANTONIO | Leonato's brother | ANTONIO | fashion-monging | 15 |
| | | | ANTONIO | thick-pleached | 14 |
| | BALTHASAR | attendant on Don Pedro | | | |
| | CONRADE | follower of Don John | | | |
| ⊟ | BORACHIO | follower of Don John | BORACHIO | five-and-thirty | 15 |
| | | | BORACHIO | church-window | 13 |
| | | | BORACHIO | chamber-window | 14 |
| | | | BORACHIO | congregation | 12 |
| | | | BORACHIO | unseasonable | 12 |
| | | | BORACHIO | Claudio--whose | 14 |
| | | | BORACHIO | contaminated | 12 |
| | | | BORACHIO | chamber-window | 14 |
| | | | BORACHIO | wedding--for | 12 |
| | | | BORACHIO | intelligence | 12 |
| ⊟ | | | FRIAR FRAN | moving-delicate | 15 |

*Figure 8-6. Relation-based hierarchy in an HFlexGrid*

## *Hierarchical Recordset Navigation*

Here's the code for the first technique, which relies on the knowledge that the "Words" column in rs is a Chapter-type Field providing filtered access to the child Recordset:

```
Public Sub printHRS(rs As Recordset)
If Not rs.EOF Then Set rsChild = rs("Words").Value
While Not rs.EOF
  Print rs("part"), rs("Description")
  While Not rsChild.EOF
     Print vbTab, rsChild("word"), rsChild("wordLength")
     rsChild.MoveNext
  Wend
  rs.MoveNext
Wend
End Sub
```

get hold of the child Recordset from the Chapter-type Field called Words

Here are selected highlights from its output:

```
DON PEDRO                  Tarragon
              Transgression                13
              pleasant-spirited            17
              unhopefullest                13
              . . .
              chamber-window               14
DON JOHN                   Don Pedro's bastard brother
              Marriage--surely             16
              circumstances                13
              chamber-window               14
              misgovernment                13
              plain-dealing                13
              enfranchised                 12
. . .
ANTONIO                    Leonato's brother
              thick-pleached               14
              fashion-monging              15
BALTHASAR                  attendant on Don Pedro
CONRADE                    follower of Don John
BORACHIO                   follower of Don Corleone
              Unseasonable                 12
. . . etc
```

Note that the `rsChild` variable was assigned only once. It would have been possible to set it inside the loop for each record, but it was done outside the loop to emphasize the way that chapters work. There is only one child Recordset, and its internal filter is updated automatically whenever the parent Recordset's current record changes. This process makes sure that the correct chapter for the current parent record is always exposed.

rs.StayInSync

While the process of updating the internal filter is automatic, it isn't forced upon you. You can switch this process off by setting the `rs.StayInSync` property to `False`. If `rs.StayInSync` is `True`, then any references you obtain to children of rs will be kept synchronized as rs is navigated. If `rs.StayInSync` is `False`, then any references to child Recordsets will keep the internal filter that applied when the reference was obtained.[4]

This means that if the rs.StayInSync setting is important to you, you should set it before obtaining any child references. By default, `rs.StayInSync`

4. It may help to think of the child reference you obtain as being a clone of the child Recordset maintained by the Chapter-type Field. If `rs.StayInSync` is `True`, rs keeps the Filter property on your child Recordset updated as rs is navigated. If `rs.StayInSync` is `False`, it leaves the clone alone (isn't that the name of a movie?).

is `True`, and this is almost always how you'll want it. However, there are occasions when you'll want the child records you are looking at to stay constant, regardless of what is happening to the parent.

As an example of using rs.StayInSync, if I introduce

```
rs.StayInSync = False
```

as the first line in printHRS (before the rsChild reference is obtained), it prints the following:

```
DON PEDRO                   Prince of Arragon
              Transgression                 13
              pleasant-spirited             17
              unhopefullest                 13
              . . .
              chamber-window                14
DON JOHN                    Don Pedro's bastard brother
. . .
ANTONIO                     Leonato's brother
BALTHASAR                   attendant on Don Pedro
. . . etc
```

At first sight, this doesn't make sense, as it looks like all the child records have disappeared, except for DON PEDRO's. You were probably expecting to see DON PEDRO's big words appear after each parent record. If you were, your thinking was correct, but referring back to the code will pay dividends. With `rs.StayInSync` set to `False`, moving on to DON JOHN has no effect on rsChild—and that is exactly the point. The cursor is still at the end of the Recordset, and so no further child record printing will occur before a Move* operation on rsChild resets the Recordset cursor.

## *Generic Hierarchical Recordset Navigation*

In about as many lines as it took to write printHRS, it's possible to write a completely generic procedure that will print any hierarchical Recordset, regardless of the number and arrangement of children. This code relies on identifying the adChapter data type and calling itself recursively when it finds a chapter. Here it is:

```
Public Sub printHRS1(    rs As Recordset, _
                         Optional iLevel As Integer)
Dim fd As Field

While Not rs.EOF
  If iLevel > 0 Then Print String(iLevel, vbTab),
  For Each fd In rs.Fields
   If fd.Type = adChapter Then
      Print
      printHRS1 fd.Value, iLevel + 1
    Else
      Print fd.Value,
   End If
  Next
  Print
  rs.MoveNext
Wend
End Sub
```

print an indent based on the depth (level) of rs in the hierarchy

printHRS1 calls itself recursively when a Chapter-type Field is identified, passing in the child Recordset and incrementing the level

Recursion was made for traversing generalized tree structures, and it's completely at home with hierarchical Recordsets. Note that setting `rs.StayInSync` won't affect this procedure, as a new child reference is acquired with a new filter setting each time it's needed.

## *Updating Hierarchical Recordsets*

As long as you remember that MSDataShape creates separate Recordsets for each SQL statement embedded within the SHAPE command, and that all Recordsets built using MSDataShape have client-side cursors, then it's fairly easy to understand the mechanics of inserts, deletes, and updates performed through hierarchical Recordsets.

However deep you are in a Recordset hierarchy, you are always operating on a straightforward Recordset. Your main responsibility is to make sure you create the Recordset using an optimistic or batch optimistic lock type. However, the following observations apply:

1. Calling transactional methods on the MSDataShape Provider results in the method calls being passed on to the Data Provider—so it's business as usual with transactions.

2. rs.Update statements apply individually to each internal Recordset.

3. rs.UpdateBatch statements also apply individually to each internal Recordset. To cause all internal Recordsets to be batch updated together, it's necessary to visit each Recordset in turn and call rs.UpdateBatch. This is readily achieved using a recursive function called within a transaction. We've already seen how recursive code can traverse through a hierarchical Recordset. The AffectRecords argument of rs.UpdateBatch can take a value of adAffectAllChapters, and it's important to use this value when batch updating child Recordsets. If you don't, then only records in the current chapter (that is, those identified by the internal filter) will be updated.

4. If the hierarchical Recordset has been disconnected (for example, if it has been marshalled between a client and server process), each internal Recordset should be separately reconnected to MSDataShape with the required Data Provider settings, before attempting any updates.[5] When disconnecting a hierarchical Recordset, you need to disconnect each Recordset individually.

Let's look at an example. You may have noticed from the results shown previously that DON JOHN's longest word ("marriage--surely") is a bit of a cheat—it's really two words joined together by a pair of dashes. You may also have noticed that his relationship with DON PEDRO is defined in rather stark terms. The following code addresses both these points by making changes at two levels in the hierarchy using batch updating:

```
Dim sCommand As String
Dim rs As New Recordset
Dim rsChild As Recordset
Dim cn As New Connection

                                                    Same command as before.
cn.Open "File Name=c:\shape.udl"
sCommand = "SHAPE {SELECT * FROM Parts} " & _
               "APPEND ({SELECT part, word, wordLength " & _
                         "FROM Words WHERE  wordLength > 11 }  " & _
               "RELATE part TO part) As Words "

      n sCommand, cn, , adLockBatchOptimistic   <-- Set the LockType.
      Child = rs("Words").Value
      d "part = 'DON JOHN'"   <-- Locate DON JOHN in the parent. Doing so sets the Chapter in the child so that the offending word can be identified.
      rsChild.Find "word like 'marr*'"
```

5. It's currently possible to perform updates with only the parent Recordset connected, but this is acknowledged as a bug by Microsoft, and you should not rely on it.

```
cn.BeginTrans
rs!Description = "Don Pedro's illegitimate brother"
rsChild!word = "surely"
rsChild!wordLength = 6
rs.UpdateBatch                                    Perform the batch updates
rsChild.UpdateBatch adAffectAllChapters           and commit.
cn.CommitTrans
```

This example also shows how to use batch updating on a connected Recordset. Even if disconnected Recordsets aren't appropriate, batch updating provides a convenient way to ensure that all changes made by a user are handled as part of the same transaction, without needing to keep a transaction open for a long period of time.

I used an ODBC-based connection and traced the activity. Here's the SQL generated by the two batch statements:

```
UPDATE "Parts" SET "description"=?
        WHERE "part"=? AND "description"=?
UPDATE "Words" SET "word"=?,"wordLength"=?
        WHERE "word"=? AND "wordLength"=? AND "id"=?
```

This is business as usual for client-side cursor updates. Note that for the child Recordset, SQLServer has silently included the primary key as a hidden column so that it can be used to generate updates correctly. As we have previously discussed, if your Provider doesn't support hidden columns, you'll need to make sure the primary key is part of the Recordset for updates to work effectively.

## *Creating Complex Shapes Using Reshaping*

MSDataShape remembers that each parent and child Recordset that is created on a live Connection object. This means that if you give a name or alias to the Recordset inside the SHAPE command string, you can reuse that Recordset by referring to it by name in a later SHAPE command. This has benefits for both performance and complexity management. This process of reusing a previously shaped Recordset in a new SHAPE command is called *Reshaping*.

You can also programmatically identify a shaped Recordset by examining its Reshape Name dynamic property. You can't write to this property directly because it is under the control of MSDataShape, but it can be used as you traverse a hierarchical Recordset to identify a particular child by name.

To explore these ideas with a more complex hierarchy, consider a hierarchical Recordset with the following overall structure:

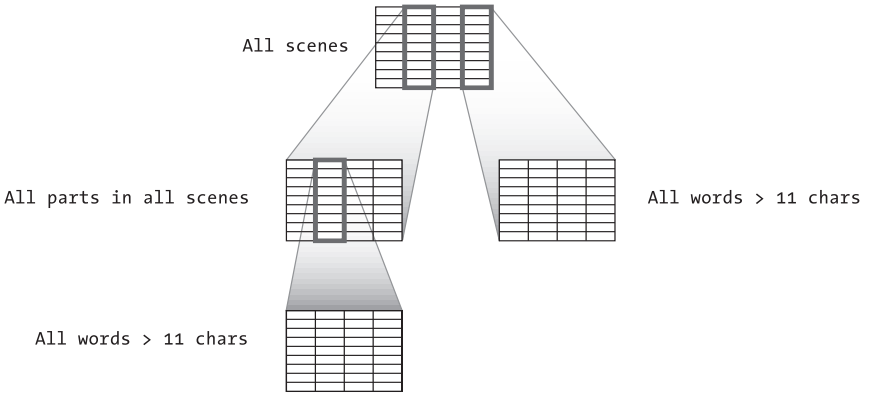



*Figure 8-7: A hierarchical Recordset before Reshaping*

This hierarchy will allow you to start looking at scenes. You can either drill down and see all the parts in each scene and then drill down again to see all the big words for each part (by part, rather than scene), or look at all the big words organized by scene.

This stretches the SHAPE language a good bit further than we have done so far, as it involves appending two children to a parent, and appending a grandchild to one of those children. The resulting SHAPE command does look complex, but it follows a regular syntax pattern. Also, we'll see how using Naming and Reshaping allows us to manage the complexity.

Here's the SHAPE command:

```
SHAPE {SELECT * FROM Scenes}
   APPEND
    ((SHAPE{SELECT DISTINCT P.part, W.act, W.Scene
         FROM Parts P, Words W WHERE P.part = W.part}
           APPEND ({SELECT * FROM Words WHERE worldLength > 11}
                 RELATE part to part))
         RELATE act to act, scene to scene),
    ({SELECT * FROM Words WHERE  wordLength > 11}
         RELATE act to act, scene to scene)
```

children to append

grandchild to append

And here's the single SQL statement that results from executing the preceding code:

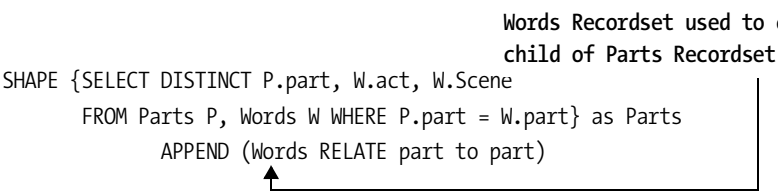
```
SELECT * FROM Scenes;SELECT DISTINCT P.part, W.act, W.Scene FROM Parts P,
Words W WHERE P.part = W.part;SELECT * FROM Words WHERE wordLength >
11;SELECT * FROM Words WHERE wordLength > 11
```

This compound string is passed as a single request to the Provider. There are two things to note about it. The first is that it contains a join. Joins and shaping can be mixed. The second thing to note is that exactly the same query was executed twice. This is wasteful, and we'll see shortly how to prevent it from happening.

This command becomes much less scary if we build it up in sections. We can start with the query that gets all the big words:

```
SHAPE {SELECT * FROM Words WHERE wordLength > 11} as Words
```

We can create this query as a standalone shaped Recordset. Because we have named the record that this command will generate (we called it Words), we can use its name in place of an embedded command when creating the child Recordset for the "parts in scenes" query:




We've also named this hierarchical Recordset, calling it Parts. We can use both of these names to create the complete four-Recordset command:

```
SHAPE {SELECT * FROM Scenes}
       APPEND
           (Parts RELATE act to act, scene to scene),
           (Words RELATE act to act, scene to scene)
```

All of this is very unscary indeed. Let's take a look at the complete program that creates this hierarchy:

```
Dim cn As New Connection
Dim rsWords As New Recordset
Dim rsParts As New Recordset
Dim rs As New Recordset
Dim sCommand As String

cn.Open "File Name=c:\Shape.udl"
rsWords.Open "SHAPE {SELECT * FROM Words " & _
                 "WHERE  wordLength > 11} as Words", cn
rsParts.Open "SHAPE {SELECT DISTINCT P.part, W.act, W.Scene " & _
               "FROM Parts P, Words W " & _
                   "WHERE P.part = W.part}as Parts " & _
                   "APPEND (Words RELATE part to part) ", cn
```

```
sCommand = "SHAPE {SELECT * FROM Scenes}  " & _
            "APPEND (Parts RELATE act to act, scene to scene)," & _
            "(Words RELATE act to act, scene to scene)"

rs.Open sCommand, cn
```

Now that we've created this complex hierarchy, it's time to do some things with it. For example, we can traverse through all its individual Recordsets and print out their names and record counts using the following recursive program:

```
Public Sub traverse(rs As Recordset, Optional iLevel As Integer)
Dim fd As Field
If iLevel > 0 Then Print String(iLevel, vbTab),
Print rs.Properties("Reshape Name"), rs.RecordCount
For Each fd In rs.Fields
  If fd.Type = adChapter Then
       traverse rs(fd.Name).Value, iLevel + 1
  End If
Next
End Sub
```

With the exception of the parent Recordset, the output of this routine will vary depending on the current record, because the current record determines the size of each chapter in a chapter field. Here's the output of passing `rs` to `traverse`, with each internal Recordset pointing at its first record:

```
DSRowset1      16
     Parts          8
          Words          7
     Words          7
```

It's pure coincidence that both uses of the Words Recordset report the same record count. As proof, here's the printout after moving to the third row:

```
DSRowset1      16
     Parts          12
          Words          19
     Words          18
```

What's interesting is that there are only three physical Recordset structures created for this command. The following listing shows the SQL generated

when executing the preceding code. Each statement was executed separately by MSDataShape.

```
SELECT * FROM Words WHERE  wordLength > 11
SELECT DISTINCT P.part, W.act, W.Scene
       FROM Parts P, Words W WHERE P.part = W.part
SELECT * FROM Scenes
```

The SQL shows us that there is only one physical Recordset providing the Word data. But the hierarchy contains Words Recordsets that have different record counts. How do we make sense of this? Once again, what we've previously learned about Clones may help here. We know that it's possible to maintain two different views of the same physical data using Clones, which is exactly what is happening in this hierarchical Recordset, where we have two separate representations of the same Words data.

Reshaping is a great way of simplifying the construction of complex hierarchies. However, the following limitations should be borne in mind:

- You have to keep the Recordsets and the Connection open if MSDataShape is to remember existing Recordset names.
- You can't use one Connection to reshape Recordsets created with another.
- You can't Append children onto an existing shaped Recordset, so you can't reuse any Recordset as a parent.
- You can't Reshape parameterized Recordsets.

## *Chapterless Child Recordsets*

There are times when you might want to perform an operation on all chapters within a child Recordset. This can easily be achieved by Reshaping a named Recordset. For example, this code:

```
Dim rsChild As New Recordset
rsChild.Open "SHAPE Words", cn
```

will extract a child Recordset named Words from a hierarchical Recordset, and provide access to all its records.

### *Avoiding Command Objects*

Unless instructed otherwise, MSDataShape uses Command objects to execute the commands within curly brackets. If you are working with a Provider that doesn't support Command objects, you can avoid them by using the SHAPE language's TABLE keyword. For example, you can replace this statement:

```
SHAPE {SELECT * FROM Scenes}
```

with this one:

```
SHAPE TABLE Scenes
```

This approach can be useful with Simple Providers (see Chapter 11), which don't use Command objects.

#### In This Section

We looked at relation-based hierarchies in depth. We saw how to use the SHAPE language to create hierarchical Recordsets, and various approaches for navigating them, including the use of recursion. We also explored how to modify hierarchical Recordsets, which can be done at any level in the hierarchy. Finally, we looked at Reshaping and Naming and how these can make our code less complex and more efficient.

## Extending and Fabricating Recordsets Using Data Shaping

Once you start becoming comfortable with the basic idea of Data Shaping, some interesting possibilities become apparent. For example, we've seen that the APPEND statement appends new Fields to an existing Recordset. All the Fields we have appended so far have been of type adChapter. However, if we can append one type of Field, then why not another?

There are many times when I want to add Fields to existing Recordsets so that I can add data that may not be in the database and continue to exploit the convenience of working with a single Recordset. While you can do a hatchet job on SQL to simulate additional Fields, they rarely work out the way you want them to, and you know in the back of your mind that what you have done is a bit of a hack.[6]

6. It's been fortunate for my career that the back of my mind is not especially smart.

The SHAPE language provides an elegant way to extend existing Recordsets. For example, consider the following syntax:

```
SHAPE {SELECT * FROM Parts}
       APPEND NEW adChar(1) AS sex
```

If this were legal, you would think that this command would create a Recordset based on the Parts table, and extend it by adding a Field called sex. It's legal. You can do it.

The following function returns a disconnected Recordset with a sex Field added to Parts and populated with "F"s or "M"s.

```
Public Function getExtendedParts() As Recordset
Dim rs As New Recordset
Dim cn As New Connection
Dim sCommand As String

cn.Open "File Name=c:\Shape.udl"
sCommand = "SHAPE {SELECT * FROM Parts}  " & _
               "APPEND NEW adChar(1) AS sex"

rs.Open sCommand, cn, , adLockOptimistic
While Not rs.EOF
  Select Case Trim(rs!part)
   Case "BEATRICE", "HERO", "MARGARET", "URSULA"
    rs!sex = "F"
   Case Else
    rs!sex = "M"
  End Select
  rs.MoveNext
  Wend
rs.Sort = "sex"                    <-- sort the Recordset using the new Field
Set rs.ActiveConnection = Nothing
Set getExtendedParts = rs
End Function
```

Note that I had to make the Recordset updateable in order to be able to populate the new column. To return a read-only Recordset, I would need to make a read-only Clone.

The following code iterates through this Recordset:

```
Dim rs As Recordset
Set rs = getExtendedParts

While Not rs.EOF
  Print rs!part, rs!sex, rs!Description
  rs.MoveNext
Wend
```

and it prints this:

```
  HERO              F            daughter to Leonato
  BEATRICE          F            niece to Leonato
  MARGARET          F            gentlewoman attending on Hero
  URSULA            F            gentlewoman attending on Hero
  DON PEDRO         M            prince of Arragon
  DON JOHN          M            Don Pedro's illegitimate brother
  . . .
```

This approach isn't without its limitations. Ideally you would have control over Field attributes and could append populated columns of standard data types to read-only Recordsets, just as you can with chapters. This isn't a replacement to writing your own Providers. However, it's an easy and elegant way to extend an existing Recordset.

## *Fabricated Hierarchical Recordsets*

You don't have to use a Data Provider with MSDataShape. Instead you can create wholly fabricated Recordsets using the same syntax you have just seen for appending new Fields to existing Recordsets, and set the `Data Provider` to `NONE`.

In the following example, I decided to enhance my reputation as a stage director and try a Shakespearean tragedy. Not having a handy database, I chose to fabricate a hierarchical Recordset with a parent containing Scene information and a child containing Parts in Scene. Here's the code for creating and populating this Recordset:

```
Dim cn As New Connection
Dim rs As New Recordset
Dim rsChild As Recordset
Dim sCommand As String
Dim vScene As Variant
Dim vPart As Variant
cn.Open "Provider=MSDataShape;Data Provider=NONE;"
```

**set the** Data Provider **to** NONE

```
sCommand = _
        "SHAPE APPEND NEW adTinyInt AS act," & _
            " NEW adTinyInt AS scene," & _
            " NEW adVarChar(50) AS description," & _
        " ((SHAPE APPEND NEW adChar(20) AS part," & _
                    " NEW adVarChar(50) AS description," & _
                    " NEW adTinyInt AS act," & _
                    " NEW adTinyInt AS scene) " & _
                " RELATE act TO act, scene TO scene) as Parts"

rs.Open sCommand, cn, , adLockBatchOptimistic

vScene = Array("act", "scene", "description")
vPart = Array("part", "description", "act", "scene")

rs.AddNew vScene, Array(1, 1, "An Open Place")
Set rsChild = rs("Parts").Value
With rsChild
 .AddNew vPart, Array("Witch 1", "First Witch", 1, 1)
 .AddNew vPart, Array("Witch 2", "Second Witch", 1, 1)
 .AddNew vPart, Array("Witch 3", "Third Witch", 1, 1)
End With

rs.AddNew vScene, Array(1, 2, "A Camp")
With rsChild
 .AddNew vPart, Array("Duncan", "King of Scotland", 1, 2)
 .AddNew vPart, Array("Malcolm", "Duncan's son", 1, 2)
End With
Set HFlex.Recordset = rs
```

create the parent Recordset

append a child Recordset

parent requires a record before you can access a Field

which will display

Grid Display of Hierarchical Recordset

| | act | scene | description | part | description | act | scene |
|---|---|---|---|---|---|---|---|
| ⊟ | 1 | 1 | An Open Place | Witch 1 | First Witch | 1 | 1 |
| | | | | Witch 2 | Second Witch | 1 | 1 |
| | | | | Witch 3 | Third Witch | 1 | 1 |
| ⊟ | 1 | 2 | A Camp | Duncan | King of Scotland | 1 | 2 |
| | | | | Malcolm | Duncan's son | 1 | 2 |

*Figure 8-8. Fabricated hierarchy in an HFlexGrid*

Once you have fabricated a hierarchical Recordset, you can do the usual kinds of things with it.

## Combining Provider and Fabricated Recordsets

Finally, you can mix Provider-generated and fabricated Recordsets in the same hierarchical Recordset. We'll see this technique in the following code. I need to start casting for *Much ADO about Nothing*, and I have an external source of information about actors who may be suitable for certain parts. I can insert this data as a fabricated Recordset, created as a child of a Recordset based on the Parts table. Here's some code to do this:

```
Dim rs As New Recordset
Dim rsChild As Recordset
Dim cn As New Connection
Dim sCommand As String
Dim vActors As Variant

cn.Open "File Name=c:\Shape.udl"
sCommand = "SHAPE {SELECT * FROM Parts} " & _
           "APPEND ((SHAPE APPEND NEW adChar(20) AS part" & _
                       " NEW adVarChar(50) AS actor," & _
                       " NEW adBoolean AS available," & _
                       " NEW adChar(6) AS price)" & _
                 " RELATE part To part) as actors"
vActors = Array("part", "actor", "available", "price")

rs.Open sCommand, cn, , adLockOptimistic
Set rs.ActiveConnection = Nothing
Set rsChild = rs("actors").Value
With rsChild
  .AddNew vActors, Array("DON PEDRO", "Tom Hanks", False, "high")
  .AddNew vActors, Array("DON PEDRO", "Gary Cornell", True, "low")
  .AddNew vActors, Array("DON JOHN", "Russ Lewis", True, "free")
End With
```

append a fabricated Recordset to one retrieved from the MuchADO database

Here's what it looks like in an HFlexGrid:

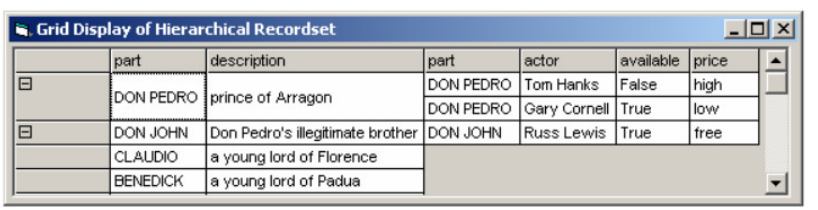



*Figure 8-9: A fabricated child of a Provider-generated parent*

In This Section

The SHAPE language can add Fields of any data type (not just adChapter) to an existing Recordset, which has some useful data management opportunities. We can take this idea to its logical conclusion and fabricate entire Recordsets, which can be mixed with Provider-generated Recordsets if required.

## Parameterized Data Shaping

There are two ways of using parameters with Data Shaping:

- **External parameters**: These make use of ADO Command and Parameter objects and work in exactly the same way as parameters do when working directly with a Data Provider. As long as you don't mix external and internal parameters, you can include external parameters at any level in a SHAPE command hierarchy.

- **Internal parameters**: These are used by the SHAPE language internally and don't require or use ADO Parameter objects. They are what most developers are referring to when discussing parameterized Data Shaping. Their purpose is to allow child Recordsets to be constructed, one chapter at a time, on an as-needed basis. This can have major performance implications. For example, if a parent Recordset has one hundred records and the typical user only wants to drill down to half a dozen at any one time, the total amount of data retrieved will be far less when data is fetched one chapter at a time. You cannot disconnect or marshall internally parameterized hierarchical Recordsets.

The two types can be mixed, although as we shall see, certain rules apply if you want to mix external and internal parameters and stay sane.

### *External Parameters*

Because we previously discussed parameterized ADO commands at some length, there is little that needs to be added about using external parameters with Data Shaping. A simple example will show you all you need to know:

```
Dim sCommand As String
Dim rs As New Recordset
Dim cd As New Command
```

```
sCommand = "SHAPE {SELECT * FROM Parts WHERE part > ? } " & _
               "APPEND ({SELECT part, word, wordLength " & _
                         "FROM Words WHERE  wordLength > ? }  " & _
               "RELATE part to part)"
cd.CommandText = sCommand
cd.ActiveConnection = "File Name=c:\shape.udl"
Set rs = cd.Execute(, Array("L", 12))
printHRS1 rs
```

**pass two parameters to the command, using standard ADO techniques**

Here, we are using parameter place markers to control which parts we retrieve in the parent, and the size of words required for the child. Note that the construction of the child Recordset is independent of how many parts are retrieved by the parent. This is potentially wasteful. You'll see that there are ways to address this shortcoming—the simplest is to add another subclause (based on part) to the child's WHERE clause and to use another parameter for its value.

The preceding code uses our generic print routine to print the results, which look like this:

```
LEONATO        governor of Messina
          LEONATO           candle-wasters       14
          LEONATO           advertisement        13
LORD           Null
MARGARET       gentlewoman attending on Hero
          MARGARET          ill-qualities        13
MESSENGER      Null
. . .
```

## *Internal Parameters*

Internal parameters cause the MSDataShape to behave very differently from external parameters. Internal parameters are defined by using the PARAMETER keyword in the RELATE clause of a SHAPE command. Here's an example, based on the simple SHAPE command we've been using all along:

```
Dim sCommand As String
Dim rs As New Recordset
```

```
sCommand = "SHAPE {SELECT * FROM Parts}" & _
               "APPEND ({SELECT part, word, wordLength " & _
                     "FROM Words WHERE  wordLength > 11 " & _
                     " and part = ? }" & _
               "RELATE part TO PARAMETER 0 ) as Words"

rs.Open sCommand, "File Name=c:\shape.udl"
```

create an internal parameter

The only SQL that is *executed* when rs is opened is this:

```
SELECT * FROM Parts
```

However, MSDataShape also *prepares*[7] the following statement:

```
SELECT part, word, wordLength FROM Words
WHERE  wordLength > 11  and part = ?
```

As you might expect, opening rs with an internally parameterized command will be much faster than opening it with a standard relation-based command, because no Words data needs to be retrieved.

The first time any Chapter-type Field's data is accessed[8], MSDataShape executes the child command, substituting the parameter marker with the value of the related field in the current record of the parent. For example, this line:

```
Print rs("part"), rs("Words").Value.RecordCount
```

will fetch the nine child records associated with DON PEDRO, and then print

```
DON PEDRO                    9
```

These records are added into the child Recordset. When a new parent record is visited and the Words field accessed, the prepared statement will be executed again, and the retrieved records will be added into the same child Recordset.

7. The difference between executing and preparing a statement was discussed in Chapter 5. Preparing a statement asks the data source to get ready to execute, but returns no data.

8. Simply visiting the Field won't trigger the retrieval of the chapter's data. That will only happen when chapter data is read.

This is a very neat way of limiting the amount of data retrieved, and it's almost essential when creating a very large hierarchy with many drill-down options and many records at each level. However, note that although it's faster at opening the Recordset, it will be less responsive once opened than a nonparameterized query, because of the need to retrieve data each time a new parent record is visited.

By default, ADO caches each chapter as it's being retrieved. This is a good default because it has performance benefits, but there may be times when you want the prepared statement to be reexecuted when you return to a previously visited record and access a Chapter-type Field. You can make this happen by setting the `Cache Child Rows` dynamic property to `False`. You have to do this before opening the Recordset, and because this dynamic property is added by the Client Cursor Engine, you have to make the Recordset use a client-side cursor explicitly, rather than rely on MSDataShape to do it for you; for example:

```
rs.CursorLocation = adUseClient
rs.Properties("Cache Child Rows") = False
rs.Open sCommand, "File Name=c:\shape.udl"
```

It's pointless using internally parameterized commands if you intend to iterate through each parent record and inspect its Chapter-type Fields as soon as the Recordset is opened. It's far, far more efficient to retrieve all the child Recordset's data in one query, than to fetch it a chapter at time. It's only when chapters are retrieved on an as-needed basis that the internally parameterized approach can be more efficient than retrieving all the data at once.

Unfortunately, the Microsoft HFlexGrid control attempts to populate the entire Recordset in order to display it, and therefore, should not be used with internally parameterized commands.

## *Combining Internal and External Parameters*

There is significant scope for confusion between ADO and MSDataShape when you try to mix internal and external parameters. However, good results can be achieved by following these two rules:

1. Never attempt to mix both types of parameter in the same internal Recordset.

2. Use external parameters in parents and internal parameters in children.

In the following code, an external parameter controls which parts are created, while an internal parameter in the child Recordset ensures that Word chapters are created only when needed:

```
Dim sCommand As String
Dim rs As New Recordset
Dim cd As New Command

sCommand = "SHAPE {SELECT * FROM Parts where part > ? } " & _
              "APPEND ({SELECT part, word, wordLength " & _
                        "FROM Words WHERE  wordLength > 11 " & _
                                    "and part = ? }  " & _
              "RELATE part to PARAMETER 0)"
cd.CommandText = sCommand
cd.ActiveConnection = "File Name=c:\shape.udl"
Set rs = cd.Execute(, 13)
```

external parameter

internal parameter

ADO sees only the external parameter.

Some interesting results can be quickly achieved by combining parameterized queries and extended Recordsets. For example, consider the user interface here:


*Figure 8-10: A simple word browser using a parameterized SHAPE command*

Users can use the Next and Previous buttons to move around an externally parameterized Parts Recordset. Each time they navigate, the form shows all words larger than wordLength for the current part. wordLength can be adjusted as the user scrolls. The form uses a standard ADO Data Grid (not an HFlexGrid) and an ADO Data Control that is hidden. Apart from that, there are two command buttons, a textbox, and a label. Here's the complete code:

```
Private rs As New Recordset

Private Sub Form_Activate()
Dim sCommand As String
Dim cd As New Command
sCommand = _
  "SHAPE {SELECT * FROM Parts where part > ? } " & _
    "APPEND " & _
       "NEW adInteger as wordLength," & _
       "({SELECT part, word, wordLength " & _
           "FROM Words WHERE  wordLength > ? and part = ? }  " & _
      "RELATE wordLength TO PARAMETER 0, part TO PARAMETER 1) As Words"
cd.CommandText = sCommand
cd.ActiveConnection = "File Name=c:\shape.udl"
cd.Parameters(0).Value = "L"
rs.Open cd, , , adLockBatchOptimistic

rs("wordLength") = CInt(txtWordLength.Text)

Set rsChild = rs("Words").Value
Set DataGrid1.DataSource = Adodc1
Set Adodc1.Recordset = rsChild
End Sub

Private Sub cmdNext_Click()
rs.MoveNext
rs("wordLength") = CInt(txtWordLength.Text)
Set Adodc1.Recordset = rs("Words").Value
End Sub

Private Sub cmdPrevious_Click()
rs.MovePrevious
rs("wordLength") = CInt(txtWordLength.Text)
Set Adodc1.Recordset = rs("Words").Value
End Sub
```

append a standard Integer Field and a Chapter-type Field to Parts

create a parameterized Command and use it as the source to rs.Open

set the value of the newly appended wordLength Field on the parent's first record, then assign the child to the grid

move to the next record, set the parent's wordLength Field from the TextBox, and rebind

The key to this example is the use of an appended Integer Field to the parent Recordset, which controls how much data the child retrieves. Rebinding the grid forces the child data to be requeried each time. Note that it was necessary to use an external parameter and to create an updateable Recordset. Calling cd.Execute doesn't return an updateable Recordset. Therefore, we applied the ADO 2.5 technique of using a Command object as the source of an rs.Open.

In This Section

Parameters can be either external to MSDataShape (in which case they are used like standard ADO Parameter objects) or internal to MSDataShape. Internal parameterized commands retrieve a child Recordset's data, chapter-by-chapter and on an as-needed basis, instead of retrieving all the child's data when the hierarchical Recordset is first created.

## Group-based Hierarchies

The basic form of a SHAPE command for creating a group-based hierarchy is

```
SHAPE {<child-command> } as Alias
        COMPUTE <parent-fields>
        BY <child-fields>
```

This is a very different structure from relation-based commands. The key point to keep in mind is that the command that appears after the `SHAPE` keyword becomes the child Recordset in a group-based hierarchy, rather than the parent. The parent is defined by the `COMPUTE` statement. This makes perfect sense, because the `COMPUTE` statement defines aggregated data based on the child Recordset, and the aggregated data appears at the top of the hierarchy, with the detail (chapters of the child Recordset) appearing below. MSDataShape builds an internal filter to create chapters in the child Recordset, and the filter is based on the Fields named in the `BY` statement.

The parent Recordset needs a Chapter-type Field in order to provide access to the child, therefore, the `COMPUTE` statement must contain the child Recordset referenced by its name or alias. This will then be split up into chapters according to the `BY` statement. The remainder of the `COMPUTE` statement contains whichever aggregates are required in the parent. Let's look at an example:

```
Dim rs As New Recordset
Dim sCommand As String
sCommand = "SHAPE " & _
     "{SELECT * FROM Words } as Words " & _
        "COMPUTE Words , SUM (Words.wordCount) As WordCount, " & _
                        " AVG (Words.wordLength) As AveLength " & _
        "BY act, scene"
```

```
rs.Open sCommand, "File Name=c:\shape.udl"

Print "Act", "Scene","Count","Ave"
While Not rs.EOF
   Print rs!act, rs!scene, rs!WordCount, _
             FormatNumber(rs!AveLength, 2)
   rs.MoveNext
Wend
```

There are three Fields defined in the `COMPUTE` statement (one is a Chapter-type Field), and these combined with the two in the `BY` statement means that the parent will have five Fields. Here's the printout as the code iterates through the parent:

| Act | Scene | Count | Ave |
|---|---|---|---|
| 1 | 1 | 2464 | 4.90 |
| 1 | 2 | 215 | 4.32 |
| 1 | 3 | 560 | 4.83 |
| 2 | 1 | 2931 | 4.82 |
| 2 | 3 | 2092 | 4.78 |
| 3 | 1 | 1327 | 5.02 |
| 3 | 2 | 918 | 4.70 |

And here's what it looks like in an HFlexGrid with some rows collapsed:

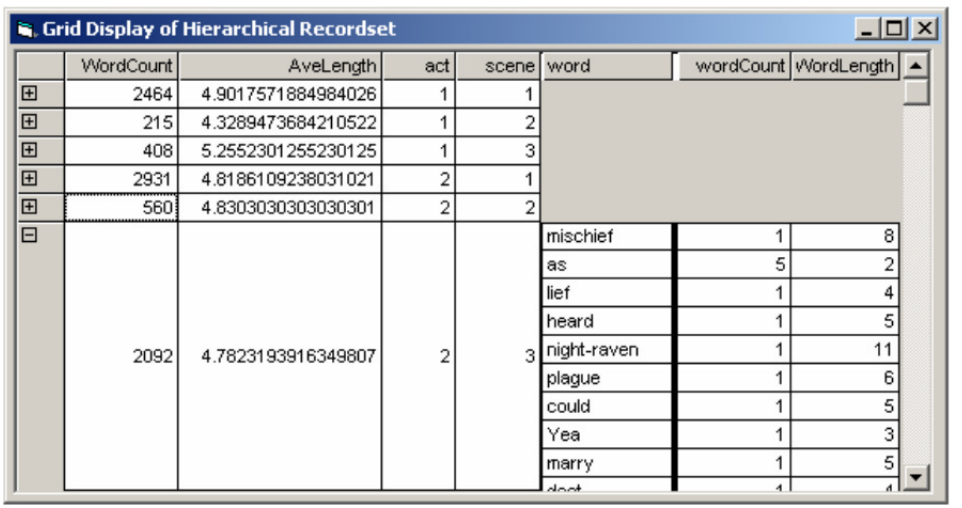



*Figure 8-11. Group-based hierarchy in an HFlexGrid*

It's possible to create complex group-based hierarchies, and once again, Reshaping and Naming can come to your rescue. For example, consider the following highly useful way of looking at the Words table:

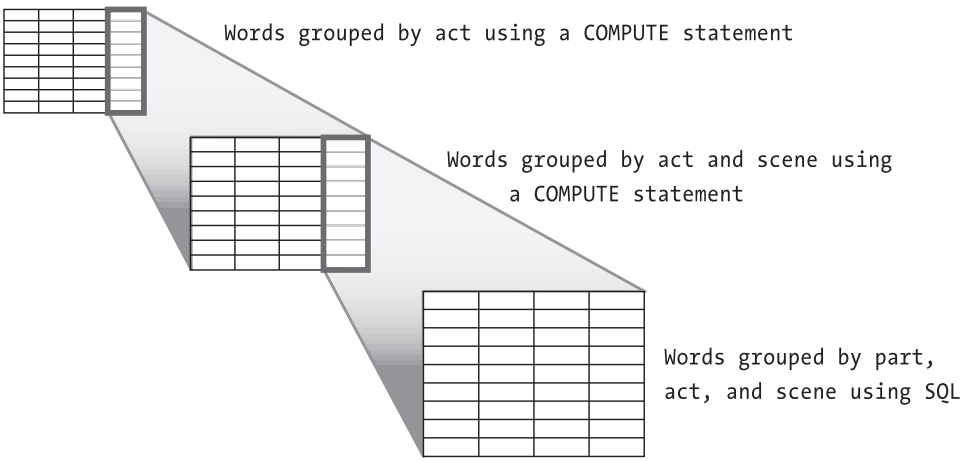



*Figure 8-12. A complex group-based hierarchy*

The following code builds up this group-based hierarchical Recordset using Reshaping to keep the complexity under control. Because the SQL for the grandchild is nontrivial, it's created first as a separate SQL statement and then concatenated into the SHAPE command for the child. The child is named, and its name is used in the parent Recordset to assemble the complete structure.

```
Dim rsChild As New Recordset
Dim rs As New Recordset
Dim cn As New Connection
Dim sCommand As String

cn.Open "File Name=c:\shape.udl"
sCommand = "SELECT  act, scene, part, " & _
             "Sum(wordCount) as PWordCount, " & _
             "Avg(CONVERT(float,wordLength)) as PAveLength " & _
           "FROM Words group by part, act, scene"

sCommand = "SHAPE(SHAPE {" & sCommand & " } as rs " & _
            "COMPUTE rs, " & _
             "SUM (rs.PWordCount) as SWordCount, " & _
             "AVG (rs.PAveLength) as SAveLength BY act, scene )" & _
           "as Scenes"
```

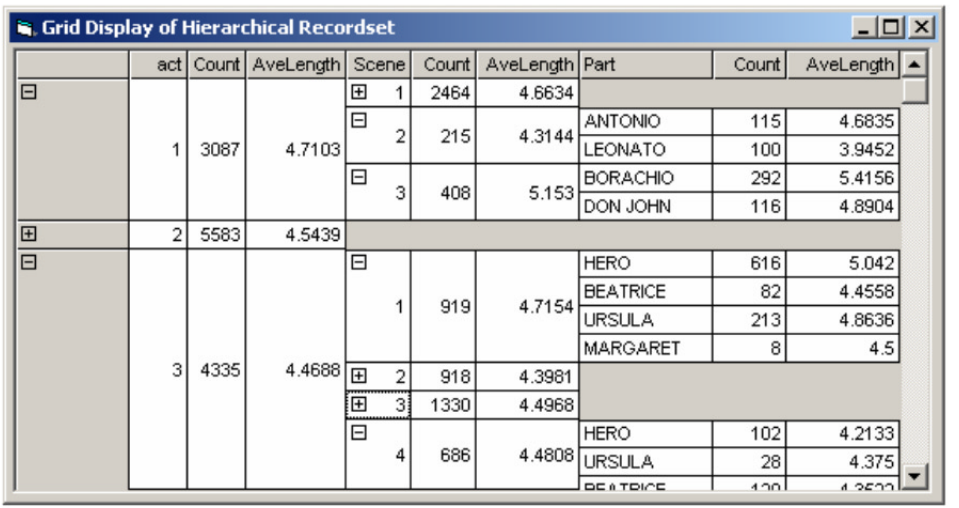
**Pure SQL creates the Parts (within a scene)-level data. wordLength is converted to a Float to get an average with decimal places.**

**Create an aggregate parent (called Scenes) based on the SQL.**

```
rsChild.Open sCommand, cn
sCommand = "SHAPE Scenes " & _
            "COMPUTE Scenes, " & _
              "SUM (Scenes.SWordCount) as AWordCount, " & _
              "AVG (Scenes.SAveLength) as AAveLength BY act"

rs.Open sCommand, cn
```

Create an aggregate parent containing act-level data using Scenes as a child.

That code creates rather a nice HFlexGrid display:




| | act | Count | AveLength | Scene | Count | AveLength | Part | Count | AveLength |
|---|---|---|---|---|---|---|---|---|---|
| ⊟ | | | | ⊞ 1 | 2464 | 4.6634 | | | |
| | 1 | 3087 | 4.7103 | ⊟ 2 | 215 | 4.3144 | ANTONIO | 115 | 4.6835 |
| | | | | | | | LEONATO | 100 | 3.9452 |
| | | | | ⊟ 3 | 408 | 5.153 | BORACHIO | 292 | 5.4156 |
| | | | | | | | DON JOHN | 116 | 4.8904 |
| ⊞ | 2 | 5583 | 4.5439 | | | | | | |
| ⊟ | | | | ⊟ | | | HERO | 616 | 5.042 |
| | | | | 1 | 919 | 4.7154 | BEATRICE | 82 | 4.4558 |
| | | | | | | | URSULA | 213 | 4.8636 |
| | 3 | 4335 | 4.4688 | | | | MARGARET | 8 | 4.5 |
| | | | | ⊞ 2 | 918 | 4.3981 | | | |
| | | | | ⊞ 3 | 1330 | 4.4968 | | | |
| | | | | ⊟ 4 | 686 | 4.4808 | HERO | 102 | 4.2133 |
| | | | | | | | URSULA | 28 | 4.375 |

*Figure 8-13. Complex group-based hierarchy in an HFlexGrid*

## *Functions Supported by the COMPUTE Statement*

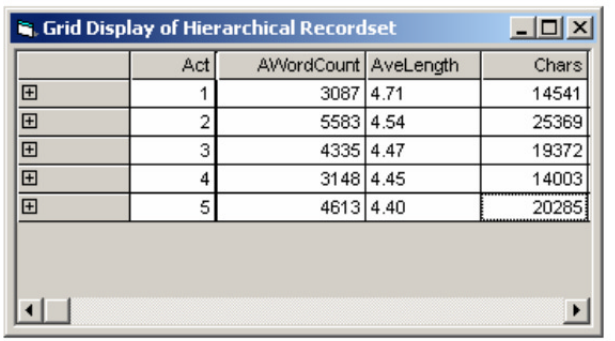
We've seen that in addition to referencing the child Recordset, the COMPUTE statement can include SUM and AVG aggregations. Here are all the aggregation functions supported by the SHAPE language:

SUM
AVG
MIN
MAX
COUNT
STDEV
ANY
CALC

All but ANY and CALC are self-explanatory. ANY allows you to include a noncalculated Field from the child as part of the Recordset. CALC allows you to apply VBA functions and calculations to any Field appearing on the COMPUTE line. For example, changing the parent Recordset's definition to

```
SHAPE Scenes
      COMPUTE Scenes,
        AVG (Scenes.SAveLength) as AAveLength,
        SUM (Scenes.SWordCount) as AWordCount,
        CALC (Format(AAveLength, '0.00')) as AveLength,
        CALC (CLng(AWordCount * AAveLength)) as Chars BY act
```

allows the number of characters in an act to be calculated, and it adds a formatted average length field. By hiding `AAveLength`, the following HFlexGrid display can be created:




| | Act | AWordCount | AveLength | Chars |
|---|---|---|---|---|
| ⊞ | 1 | 3087 | 4.71 | 14541 |
| ⊞ | 2 | 5583 | 4.54 | 25369 |
| ⊞ | 3 | 4335 | 4.47 | 19372 |
| ⊞ | 4 | 3148 | 4.45 | 14003 |
| ⊞ | 5 | 4613 | 4.40 | 20285 |

*Figure 8-14. A group-based hierarchy showing CALC Fields*

## *Combining Group-based and Parameterized Relation-based Hierarchies*

And for our last trick … we'll extend the preceding three-layer hierarchy with a fourth layer containing all the Words for each act/scene/part. Ideally, you would want this final layer to be parameterized, to avoid shipping the entire Words table across the network and building a client-side cursor on it.

Although a group-based parent Recordset can't have parameterized child Recordsets, it can have parameterized grandchild Recordsets (see Figure 8-15).

In our current example, the third layer is an SQL command, which is itself aggregated. We'll add a simple parameterized child to this command that draws records directly from the Words table.

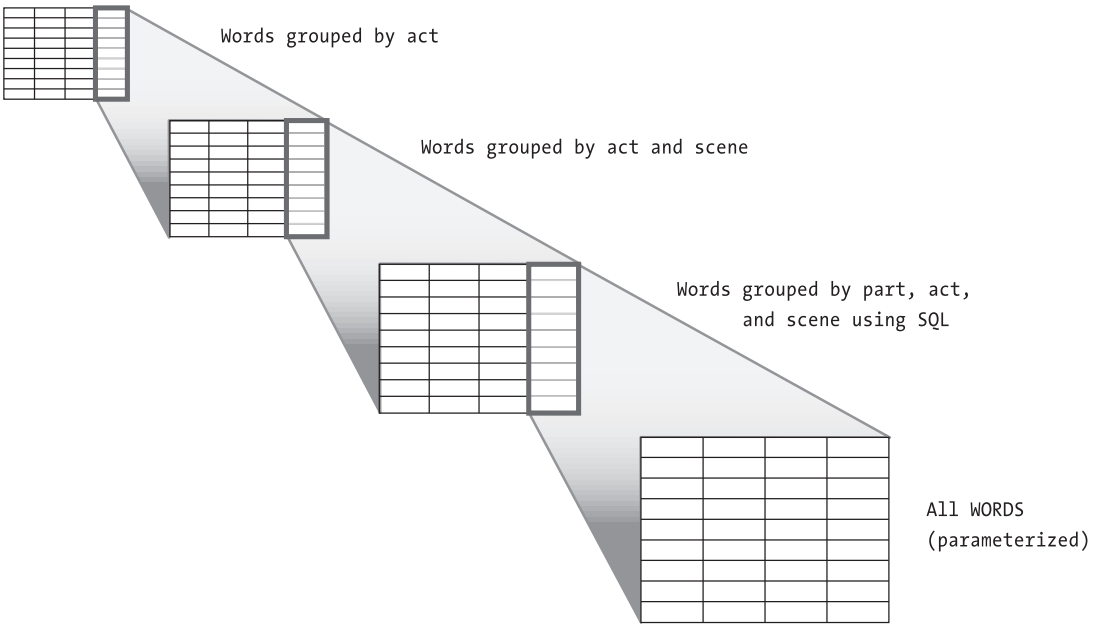



*Figure 8-15: Adding a parameterized child to the existing hierarchy*

Here's the code that creates the Scenes Recordset and assigns it to rsChild. The rest of the code from the preceding example is unchanged:

```
cn.Open "File Name=c:\shape.udl"
sCommand = _
   "(SHAPE " & _
      "{SELECT  act, scene,part, " & _                    ← the same SQL
         " sum(wordCount) as PWordCount, " & _              as before …
         " Avg(CONVERT(float,wordLength)) as PAveLength " & _
         " FROM Words group by part, act, scene} as rs  " & _
    "APPEND ({SELECT * FROM Words WHERE " & _
                "part = ? and act = ? and scene = ?} " & _
     " RELATE part TO PARAMETER 0, act TO PARAMETER 1, " & _
          "scene TO PARAMETER 2)as Words)"
```

… now with child

```
sCommand = _
  "SHAPE (SHAPE " & sCommand & "  as rs " & _
       "COMPUTE rs, " & _
          "SUM (rs.PWordCount) as SWordCount, " & _
          "AVG (rs.PAveLength) as SAveLength BY act, scene )" & _
        "as Scenes"

rsChild.Open sCommand, cn
```

The end result here is complex, but each individual step is straightforward. What makes it complex is the nature of the relationships we are modeling. You could spend a week building an application that provided users with this amount of power. Data Shaping will make your life a great deal easier.

### In This Section

We looked at group-based hierarchies. You learned how to use the COMPUTE statement to create powerful structures and displays and saw how group-based and relation-based hierarchies can be combined.

## Summary

Data Shaping is often automatically associated with the Microsoft Hierarchical FlexGrid. It's generally claimed that Data Shaping is obscure and complicated, but both of these viewpoints are mistaken.

- Data Shaping can do a great deal more than create hierarchical displays. It can extend and fabricate Recordsets and weave them into relationships that are natural for certain types of uses.

- How this data is presented is a completely separate issue, and the HFlexGrid is not always the most appropriate vehicle.

- The SHAPE language is simple, although the representations of data that it allows can be very sophisticated, resulting in complex-looking command structures. These can nearly always be broken down into more manageable chunks.

- What makes Data Shaping ultimately very accessible is that it introduces surprisingly little that is new.

Data Shaping creates standard tabular Recordsets and exploits filters and cloning to combine them in a powerful, flexible fashion, based on the concept of chapters. It provides viable alternatives, but not replacements, for certain types of more powerful SQL statements.

The key difference between Data Shaping and SQL joins and aggregation is that SQL always flattens data back into a tabular structure, whereas Data Shaping maintains or even extends the natural hierarchical relationship in the data it uses. Neither approach is best—each has its place. Deciding which to use is the primary challenge that Data Shaping introduces.

Data Shaping provides powerful new data representations, but they are still based on an essentially tabular form. In the next chapter, you'll see how ADO 2.5 allows you to leave behind the tabular model of data representation, should you so wish.