Issue Date: FoxTalk July 2000

Data Warehouses, OLAP, and You, Part 4

Leslie Koorhan
lkoorhan@earthlink.net

In his previous installment, Leslie Koorhan created an OLAP cube from the TasTrade data mart. In this series of articles, Leslie has transferred data from a relational database design to a data warehouse design and then used that as the data source for an OLAP cube. Now it's time to manage that cube through the object model that Microsoft SQL Server OLAP Services provides, Decision Support Objects. In this article, Leslie explains the Decision Support Objects object model, its collections, methods, and properties, and demonstrates the use of this model through examples.

Last month, I created an OLAP cube using the TasTrade data mart as the data source. I demonstrated the steps needed to do so, using Microsoft SQL Server OLAP Services' OLAP Manager. Using this tool, I was able to easily construct a cube using the Cube Wizard, Cube Editor, Dimension Wizard, and Storage Design Wizard. With the OLAP Manager, maintaining a cube is fairly easy and straightforward. It allows me to set up security for users to access the cubes, and also includes the ability to browse the data through a PivotTable Service.

It seems as if I can do everything with the OLAP Manager, but it might be overkill to use such a product on a daily basis. One of the really nice features of OLAP Services is the ability to manage and query the data through other programs for which I might have implemented my own interface into the cube. The reason that I'd write an application to do this is to simplify the process so that a typical user doesn't need to know how OLAP Manager works.

So, with that in mind, I'll show the object model for administration of the cube on the server side, Decision Support Objects (DSO). DSO contains special objects representing all aspects of a cube, including methods and properties to manipulate it. To illustrate the model, I've included some modest examples of code for managing the OLAP server.

Decision Support Objects model

The Decision Support Objects object model is deceptively simple. It consists of a server object that represents the OLAP Server service that you'd use, an MDStores collection that represents the objects within the server, a Dimensions collection for describing the cube, Levels collections within a dimension, and Measures collections for the facts about what's in the cube. This is deceptive because DSO is a hierarchical design in which the MDStores collection is used to describe databases, cubes, partitions, and aggregations. This means that depending on the type of MDStore object, there's another MDStores collection within it. **Figure 1** shows the DSO object model.

Figure 1:* *The Decision Support Objects (DSO) object model.

Since the MDStores collections can contain a variety of different objects, there's a property, ContainedClassType, that indicates the type of class for the objects in that particular collection. In fact, every DSO object has ClassType and SubClassType properties that describe the class of that object. In using the DSO with VisualFoxPro, I discovered that MDStores only works as a collection for the Server object. Instead, Visual FoxPro recognizes a Cubes collection for the Database object, a Partitions collection for the Cube object, and an Aggregations collection for the Partition object.

The ClassType and SubClassType properties are integer values that are referred to as *enumerations*. Enumerations are basically constants defined within a type library. If you were using Visual Basic, then you'd reference the ClassTypes enumeration to find out the different values used in the ClassType property. But instead of the actual integer value, the enumeration is used. The ClassTypes enumerations include clsServer (1), clsDatabase (2), clsCube (9), clsPartition (19), clsDatabaseDimension (7), and so on. In Visual FoxPro, you'd create an Include file with these set up as constants (#DEFINE cls_Server 1) and then use the #INCLUDE preprocessor command to include the file in your classes, forms, and programs that reference the DSO objects.

Enumerations are quite common in Visual Basic, and the OLAP documentation often refers to the enumeration name, not the actual integer value. Within this article, I'll use the enumeration name because it's self-describing.

Another commonly used term in the Visual Basic/C++/Java world is *interface*. In Visual FoxPro, interface can be thought of as the public methods, events, and properties of a class. Interface in the COM world might seem to be the same as VFP's class definition. In the DSO model, MDStore is referred to as an interface, rather than a class, as it would be in Visual FoxPro. MDStore is simply a container of multidimensional data. It can include cubes, partitions, or aggregations, depending on which type of object the container is. The ClassType property, common throughout the object model, indicates which object it is.

clsServer

The Server object, clsServer, is the main entry point for access to Microsoft SQL Server OLAP Services at runtime. The Server object allows you to access and control the OLAP server itself. This object is mainly used to establish a session with the OLAP Server through its Connect method. The only argument necessary is the name of the server itself.

```
oDSO = CREATEOBJECT("DSO.Server")
oDSO.Connect("SERVERNAME")
```

Another commonly used method is CloseServer, which ends the session and releases all resources and objects used through that server. The only other method unique to the Server object is CreateObject. By using this method, you can create any other object in the DSO model by specifying its ClassType as the first argument, and optionally the SubClassType as the second argument. Through this method, you can create an entire OLAP database, along with all of the cubes, partitions, dimensions, and measures that are a part of the OLAP database. The problem with CreateObject is that you still have to add the object to the appropriate collection (every collection has an Add method). The easier way to build the database would be to use the AddNew method of each collection. Then the object gets created and added in one command.

The most important properties of the Server object are State and ServiceState. The State property returns an enumerated value indicating the state of your connection to the OLAP Server. There are three possible values representing Connected, Failed, and Unknown. ServiceState indicates the state of the service itself, whether it's running, stopped, or paused, among others. It can also be used to change the state of the service. For example, after attempting a connection, if the State property shows the connection failed (there would also be an error), you might query ServiceState and, if the server is down, you'd try to start it and then make the connection.

```
IF oDSO.State <> 1           && stateConnected
  IF oDSO.ServiceState <> 4    && OLAP_SERVICE_RUNNING
    oDSO.ServiceState = 4
    oDSO.Connect("SERVERNAME")
  ELSE
   * report failure to make connection and end routine
  ENDIF
ENDIF
```

Common OLAP methods and properties

Because DSO has such a strong hierarchical design, there are server methods and properties common to the different objects. For example, the Update method is used to update the definition of that particular DSO object. The overall effect of this method is to update all subordinate objects as well. In the case of clsServer, that would mean all of the objects in that server. Used with an object of ClassType clsDatabase, it wouldn't update just that database, but all of the dimensions of that database as well. Definitions of objects are referred to as metadata, data about the OLAP structure itself, and are stored in an "OLAP Services repository." This repository is a relational database created automatically by OLAP Services and should be accessed only through OLAP Manager or DSO. (Actually, the database is a Microsoft Access database on the server computer.)

Because clsDatabase, clsCube, and clsPartition are all implementations of the MDStore interface, they share a number of methods as well. The most important method for MDStore is Process. This method updates the data for that particular object as well as all subordinate objects. Processing is the act of getting new, or changed, data from the data warehouse or data mart that's the data source for your OLAP database.

Just as the clsServer object has a State property, so do the MDStore interface objects. For these objects, the State property indicates various conditions depending on the ClassType. The possible values indicate whether the MDStore has never been processed, whether the structure has changed, or whether the source mapping has changed or is current. A Dimension object also has a State property.

Data updates

After all of this, you probably want to know how to update the data in the cube given changes in the source data. What kind of changes would those be? They could be just data changes. Even though a data warehouse has static data, there are two types of changes. 1) New data has been added to the warehouse. After all, you'll want the next day's, the next week's, or the next year's data at some point, and then your OLAP database needs to pick it up. 2) Oops. That's right, even with all of the careful scrubbing that your data goes through before it gets to the warehouse and the fact that you've made "sure" that only finished data goes in, something goes awry, and the data needs to be changed.

The dimensions will be changed as well. There might be new levels that you need to create or drop, new customers, new stores or dates to add, or changes to the definition of the dimensions themselves. New dimensions, new levels, changes—all of these lead to the fact that after you've made the modifications, you need to update the OLAP database.

There are basically two ways to process a cube: full or incremental. A full process is accomplished simply by running the Process method with the argument of fullProcess (an enumerated value). How OLAP accomplishes this while users are working with your cube is simple. A transaction is started, a shadow copy of the cube is created, the shadow copy is then loaded with all of the data from the source, and then after it's completed the shadow copy replaces the original cube.

A full process is very simple to implement, but it takes a long time and extra disk space, and it might not be needed if there are only incremental changes. Furthermore, although users can be using the original cube while the shadow is being built, they'll have to reconnect once the cube is replaced. There are times when only a full process can be used, such as when you create new levels within a dimension. But there's also the possibility of doing data refresh processing only. That means that only the data is read in, and this doesn't interfere with any ongoing transactions.

Another technique to consider is incremental updates. In order to use incremental updates, you'll want to break the cube into different parts, called partitions. Remember that the OLAP database contains cubes, which contain partitions, which contain the aggregations. If there's only one partition, then the whole cube has to be fully updated when there are changes to the cube structure. But by using partitions, you gain several advantages. You gain flexibility because each partition can have its own storage design. One partition can use MOLAP, which is best for heavily used aggregations; another could be ROLAP, which would be best for those aggregations that are used rarely; and other partitions could use HOLAP, where the usage is greater than ROLAP, but less than MOLAP. You also gain performance because queries are processed in parallel across partitions.

But how will you know which aggregations are used more than others? Well, OLAP Services maintains a query log of all queries made against the OLAP database. This way you can examine the log to find out which aggregations are more heavily used than others. And DSO gives you the tools necessary to do this with objects of ClassTypes clsCubeAnalyzer and clsPartitionAnalyzer. The former has only a method used to extract information from the query log, and the latter has the methods and collections that are used to design aggregations, just as the Storage Design Wizard does, when using OLAP Manager.

In the accompanying **Download file**, I show techniques for several server-side activities, including designing aggregations, updating dimensions, and usage optimization. The sample was designed as one class with several methods.

Designing aggregations

In this sample code, I duplicated the Storage Design Wizard's behavior. The method requires the name of the OLAP server; the names of the database, cube and partition to be built, or rebuilt; and the target values of the percentage benefit and maximum partition size. The last two items are used to determine when OLAP Services should stop processing. If the values are never exceeded, then all possible aggregations will be used in the partition.

I create local variables for all of the objects that I'll go through to get to the partition and the partition analyzer. First I create the DSO.Server object, and then I use the Connect method to get to the server named as a parameter. Using the MDStores collection of the Server object and the database name requested, I get a reference to the Database object. Using the Cubes collection of the Database object and the cube name, I get a reference to the Cube object. Finally, using the Partitions collection of the Cube object and the correct partition name, I get a reference to the Partition object that I want to work with.

At this point, all of the existing aggregations of the partition are removed, and I get a reference to the Partition Analyzer object. The first step in analyzing the aggregations for the partition is to run the InitializeDesign method, followed by the NextAnalysisStep method. This method requires three parameters passed in by reference so that it can return the current values of the percentage benefit, partition size, and count of aggregations built so far. This method will also return a true or false value indicating whether there are further aggregations to analyze. So I keep running this method until it returns false. But after each execution, I'll evaluate the current percentage benefit and the size of the partition to make sure that neither one exceeds the targets specified by the parameters. If either target is exceeded, then no more aggregations will be built.

After each execution of NextAnalysisStep, a new aggregation will be added to a collection of the Partition Analyzer object, DesignedAggregations. After the process of building this has completed, I then loop through the DesignedAggregations collection, adding each to the aggregation collection of the Partition object I'm working with. Then the CloseAggregationsAnalysis method is run for the Partition Analyzer, the partition is processed, and if there are no errors, it's updated and the Server object is closed.

Updating dimensions

To update a dimension requires similar information when called—that is, the name of the server, the database, and the dimension, assuming that it's a dimension at the database level. This method simply performs a refresh of the data only, not a complete rebuild of the dimension. This is done by supplying a parameter to the Process method, processRefreshData, which is a constant with a value of two.

Other methods

Included in the sample class are methods for creating new partitions and merging existing partitions. The former uses a filter to take some of the data from an existing partition and splits it off into the new one, while the merge does the opposite. The Clone and Merge methods of a Partition object are used.

There's also a method for doing usage-based optimization. This code uses the cube analyzer object's ability to read the query log to produce a recordset of datasets along with their longest times. Using this recordset, the method then checks those times against an input value. If the time is longer than that value, then it's added to the list of aggregations to be stored in the partition. This way, the most important queries will be in the cube before the partition analysis begins.

Summary

As you can see, by using Microsoft SQL Server OLAP Service's Decision Support Objects, you can manage the cube in every possible way. This object model exposes all of the functionality needed to create cubes, to analyze their usage, and to refresh them, all without having to use the OLAP Manager. This way you can write a front-end program of your own to control OLAP Services from any system on the network.

In the last installment of this series, I'll cover the other end of the cube—the data—and how you'll access it. There are two components to retrieving the data of the cube, something called MDX, and an object model called ADOMD. By learning and using these tools, you'll be able to retrieve and display the cube's data any way that you like.