WILEY

TIMELY. PRACTICAL. RELIABLE.

Data Mining with Microsoft® SQL Server® 2008

Jamie MacLennan
ZhaoHui Tang
Bogdan Crivat

CHAPTER

16

Programming SQL Server Data Mining

The concept of data mining as a platform technology opens up the doors for the possibility of a new breed of intelligent applications. An *intelligent application* is one that does not need custom code to handle various circumstances. Instead, it learns business rules directly from the data. Additionally, as business rules change, intelligent applications are updated automatically by reprocessing the models that represent the business logic.

Examples of intelligent applications are cross-sales applications that provide insightful recommendations to your users, call center applications that show only customers with a reasonable chance of making a purchase, and order-entry systems that validate data as it is entered without any custom code. These are just the tip of the iceberg. The flexibility and extensibility of the SQL Server Data Mining programming model will excite the creativity of the developer, leading to the invention of even more types of intelligent applications.

Chapter 15 explained that the core communication protocol for Analysis Services is XML for Analysis (XMLA). This protocol provides a highly flexible, platform-independent method for accessing your data mining server. Everything that can be done between the client and the server can be done through XMLA. However, just because you can do it the hard way doesn't mean that you have to.

This chapter reviews programming interfaces and object models that make it easy to write data mining applications using Analysis Services. All the examples use Visual C# .NET to demonstrate how to implement typical data mining tasks with the appropriate interface for each task, and how to use some special features of SQL Server data mining to exploit data mining programming to the fullest.

Sample code and data sets for this chapter are included in `Chapter16.zip`, which you can download from the book's companion website (`www.wiley.com/go/data_mining_SQL_2008/`). The archive contains the following:

- A SQL Server 2008 database backup containing the data sets used in this chapter
- Three projects demonstrating the different APIs presented in this chapter

In this chapter, you learn about the following:

- APIs and their application to data mining
- Using Analysis Services APIs
- Creating and managing data mining objects using Analysis Management Objects (AMO)
- Data Mining Client programming with ActiveX Data Objects (Multidimensional) (ADO MD) for .NET (ADOMD.NET)
- Writing server-side stored procedures with Server ADOMD.NET

Data Mining APIs

If you were to list the various application programming interfaces (APIs) for SQL Server Data Mining, you would get a dizzying array of acronyms. To make things even more confusing, many of the names were chosen not because of their functions, but to provide brand affinity with existing technologies. Table 16-1 describes the major APIs used in Analysis Services programming.

NOTE **See Books Online for full documentation and samples of all APIs used by Analysis Services.**

ADO

ADO was created to assist the Visual Basic programmer in accessing data residing in databases. The ADO libraries wrap the OLE DB interfaces into objects that are easier to program against. Because OLE DB for Data Mining specifies that a data mining provider must be an OLE DB provider, ADO can be used to execute data mining queries just as it does relational database queries.

Table 16-1 SQL Server Mining APIs

API	COMPLETE NAME	DESCRIPTION
OLE DB	OLE for Databases	Microsoft standard API for accessing database objects from within any application that supports COM/ActiveX technology. It is typically used in native languages such as C++. It is supported via the Microsoft OLE DB Provider for Analysis Services 10.0.
ADO	ActiveX Data Objects	Friendly and easy-to-use wrapper on top of OLE DB. It provides access to data objects (including data mining) from native languages such as Visual Basic. It works on top of any OLE DB provider, in particular on top of the Microsoft OLE DB provider for Analysis Services 10.0.
ADO.NET	ActiveX Data Objects for .NET	.NET (managed) version of the ADO library. It is a friendly and easy-to-use managed wrapper on top of OLE DB. Just like its native counterpart (ADO), it works on top of any OLE DB provider, including Microsoft OLE DB Provider for Analysis Services 10.0.
ADOMD.NET	ActiveX Data Objects (Multidimensional) for .NET	.NET (managed) dedicated provider for Analysis Services. It works only with Analysis Services and does not use OLE DB. It has the same ease of access as ADO.NET, but it is optimized for Analysis Services operations, and offers various specific classes and interfaces. It provides access to Analysis Services data objects from managed languages such as Visual Basic .NET, C#, and J#.
Server ADOMD .NET	Server ActiveX Data Objects (Multidimensional)	Provides access to Analysis Services data objects from user-defined functions running inside the server.
AMO	Analysis Management Objects	A management interface for Analysis Services that provides objects for performing operations such as creation, processing, and so on.
DMX	Data Mining Extensions	Extensions to SQL to support data mining operations.

(*continued*)

Table 16-1 (*continued*)

API	COMPLETE NAME	DESCRIPTION
OLE DB/DM	Object Linking and Embedding for Databases for Data Mining	The name of the specification that defines the DMX language. It introduces the concept of data mining models as database objects.
XMLA	XML for Analysis	A communication protocol and XML format for communicating with an analytical server independent of any platform. It is supported by Microsoft Analysis Services and constitutes the main communication protocol between any client API and the server.

ADO reduces the complexity of OLE DB interfaces to the following three essential objects:

- The *connection object* is used to connect to the server and issue schema rowset queries.
- The *command object* is used to execute DMX statements and optionally retrieve their results.
- The *record set object* contains the result of any data-returning queries.

ADO.NET

ADO.NET is the managed data access layer. It was created to allow managed languages (such as Visual Basic .NET and C#) to access data, much as ADO was created for native languages. The philosophy of ADO.NET is somewhat different from that of ADO in that ADO.NET is designed to work in a *disconnected* mode, where data can be accessed and manipulated without maintaining an active connection to the server. When work is completed, a connection can be established, and all the appropriate updates will be propagated to the server (providing that there is server support for such behavior).

ADO.NET is more modular than ADO. ADO works in one way and that way only and contains special code to interact with the SQL Server provider better than other providers. ADO.NET provides generic objects that work with any OLE DB provider and allows providers to create their own managed providers for data interaction. For example, SQLADO.NET contains objects optimized for interacting specifically with SQL Server, and similar managed providers can be written for any data source.

ADO.NET contains connection and command objects, which is similar to ADO. However, ADO.NET introduces the data set object for data interaction.

A *data set* is a cache of the server data contained in a set of data tables that can be independently updated or archived as XML. You would typically use data adapters to load data sets — either the generic adapter that is supplied with ADO.NET or a provider-specific adapter such as SQLDataAdapter. For direct data access, ADO.NET uses a data reader (similar in concept to the ADO record set) returned from its command object.

ADOMD.NET

ADOMD.NET is a managed data provider that implements the data adapter and data reader interfaces of ADO.NET specifically for Analysis Services, making it faster and more memory-efficient than the generic ADO.NET objects. In addition to the standard ADO.NET interfaces, ADOMD.NET contains data mining and OLAP-specific objects, making programming Data Mining Client applications easier.

The `MiningStructure`, `MiningModel`, and `MiningColumn` collections make it easy to extract the metadata that describes the objects on the server. The `MiningContentNode` object allows for the programmatic browsing of mining models, and can be accessed from the root of the content hierarchy or randomly from any node in the content.

NOTE There is also a native version of ADOMD.NET, appropriately named ADOMD. This interface is maintained mostly for backward compatibility with SQL Server 2000 and does not contain any objects or interfaces for data mining programming.

Server ADOMD.NET

Server ADOMD.NET is a managed object model for accessing Analysis Server objects (both data mining and OLAP) directly on the server. It is intended for use in user-defined functions and stored procedures, described later in this chapter.

AMO

AMO is the main management interface for Analysis Services. It replaces the SQL Server 2000 interface, Decision Support Objects (DSO), which is still maintained for backward compatibility but has not been updated to take advantage of all the new features of SQL Server 2005 and SQL Server 2008.

Like ADOMD.NET, AMO contains the `MiningStructures`, `MiningModels`, and `MiningColumns` collections, and the like. However, whereas ADOMD.NET is for browsing and querying, AMO is for creating and managing. You can use AMO to perform programmatically all the operations you perform in the user interfaces of Business Intelligence Development Studio (BI Dev Studio) or SQL Server Management Studio. In fact, the management operations of both user interfaces were written using AMO.

NOTE **You should use ADOMD.NET when writing Data Mining Client applications except when .NET is not available. Otherwise, use ADO (or OLE DB) for Windows applications, or plain XMLA for thin client applications. For applications in which you will be creating new models or managing existing models, use AMO. Note that AMO does not allow the execution of queries (such as prediction queries).**

Using Analysis Services APIs

Whenever you need to access any of the APIs for Analysis Services, you must ensure that you add the appropriate references to your project. Table 16-2 lists many of the APIs with the required references.

Table 16-2 Analysis Services References

API	TYPE	REFERENCES
ADO	Native	Microsoft ActiveX Data Objects
ADOMD.NET	Managed	`Microsoft.AnalysisServices.AdomdClient`
Server ADOMD.NET	Managed	`Microsoft.AnalysisServices.AdomdServer`
AMO	Managed	`Microsoft.AnalysisServices` `Microsoft.DataWarehouse.Interfaces`

To make your coding easier, add a library reference at the top of your source files so that you don't have to specify the fully qualified name for every object. For VB.NET, add the following:

```
Imports Microsoft.AnalysisServices
```

Or for C#, add the following:

```
Using Microsoft.AnalysisServices
```

Using Microsoft.AnalysisServices to Create and Manage Mining Models

In this section, you will learn how to use AMO to create and manage data mining objects (models and structures). The examples use the `MovieClick` dataset and the goal is to analyze the way different generations of customers use various channels.

If your programming interest lies only in embedding data mining into client applications, you can skip this section.

The simplest way to create mining models is to use DMX statements such as `CREATE MINING MODEL` and `INSERT INTO` with any of the command interfaces such as ADO, ADO.NET, or ADOMD.NET. Although that method has the advantage of simplicity, features such as custom column bindings and OLAP mining models (among others) are not accessible through the command-based APIs. Therefore, to ensure that your application can take advantage of all that SQL Server Data Mining has to offer, the recommended API for creating complex mining models is AMO. In fact, the creating, editing, and managing tools included in BI Dev Studio and SQL Server Management Studio were written with AMO.

Figure 16-1 shows the major AMO objects used for data mining programming. These objects will be used in the code samples throughout the AMO section of this chapter.

AMO Basics

AMO is a straightforward object model placed on top of the XML representation of Analysis Services objects. In addition to providing a convenient API, AMO provides basic validation and methods to update, change, and monitor objects on the server.

NOTE To add AMO code to your project, you must add references to two assemblies: `Microsoft.AnalysisServices` and `Microsoft.DataWarehouse .Interfaces`. Add the following line of code to the top of your source files so that you don't have to specify the fully qualified name for every object. For VB.NET, add:

```
Imports Microsoft.AnalysisServices
```

Or for C#, add:

```
Using Microsoft.AnalysisServices
```

Every object in AMO implements the `NamedComponent` interface, which supplies `Name`, `ID`, and `Description` properties as well as a `Validate` method. An object's ID is its immutable identifier that cannot be changed after it is set. This is useful when you're developing user applications with fixed objects. It allows users to arbitrarily change object names for their own needs, while providing a consistent way for your code to reference objects.

`MajorObject` inherits `NamedComponent`. `MajorObject` adds the `Update` and `Refresh` methods to update the server with local changes. `NamedComponent` refreshes the local model with the server contents. Additionally, `Major-Objects` has methods to access referring and dependent objects, and contains an `Annotations` collection for arbitrary user extensions. The `Role` object is an example of a `MajorObject`.

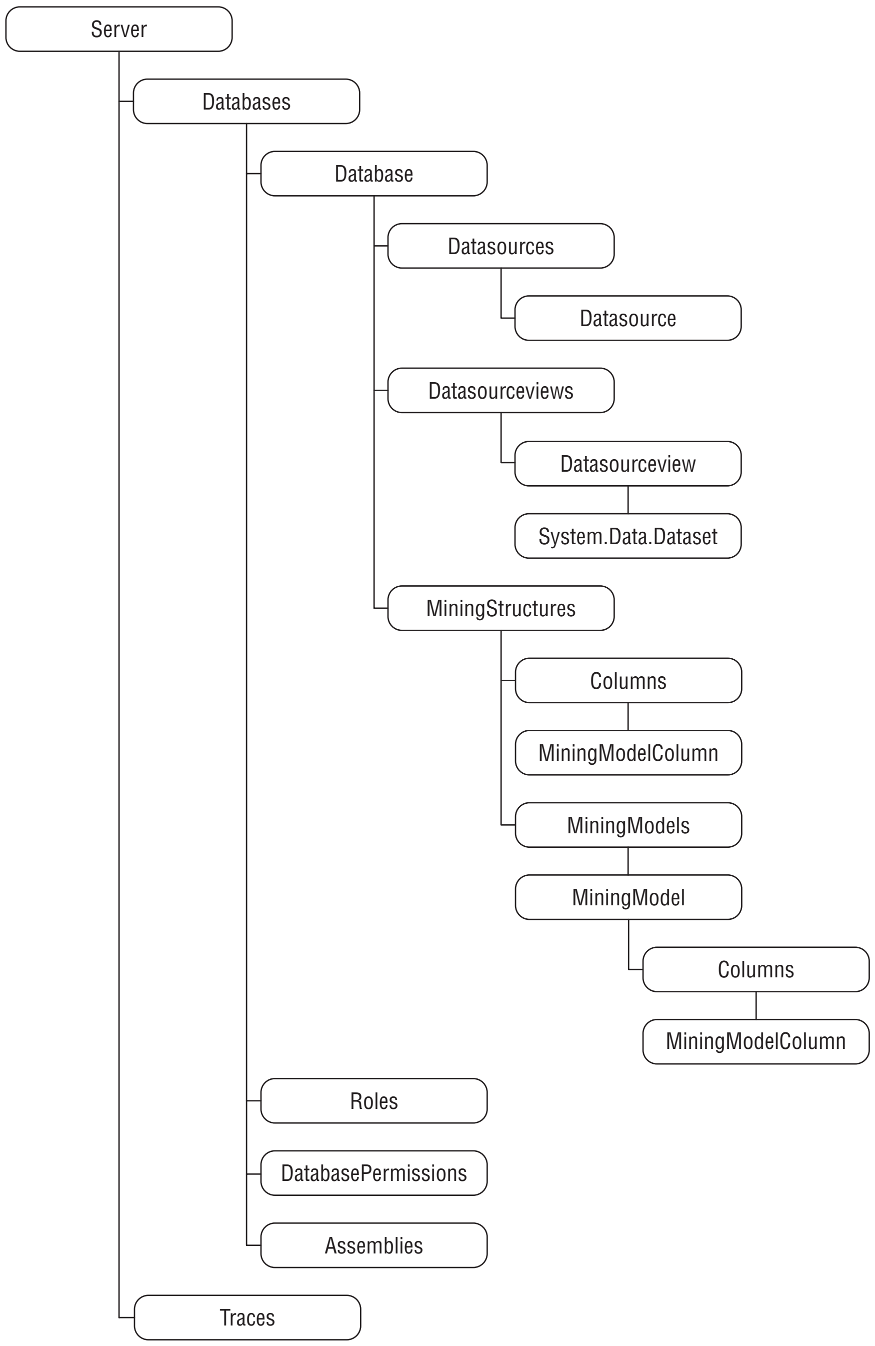

Figure 16-1 Partial AMO object hierarchy

`ProcessableMajorObject` inherits `MajorObject`, adding methods and properties to process the object and determine the processed state and last processed time. `MiningStructure` is an example of a `Processable-MajorObject`.

AMO Applications and Security

Because AMO is generally a management API, certain permissions must be present for users to use any AMO-based application. Obviously, any user with administrative permissions (members of the server Administrator role) will have access through AMO, but users with more restrictive permissions can also have limited access.

NOTE To perform certain operations, such as iterating objects, using AMO, you may need a higher level of permission than when using a command API, such as ADOMD.NET. This is because ADOMD.NET and other APIs use database schemas to access objects, rather than metadata definitions.

Table 16-3 describes the permissions necessary for a user to perform any function through AMO.

Table 16-3 AMO Permissions

FUNCTION TO PERFORM	PERMISSION REQUIRED
Iterate objects	Access and Read Definition
View object definitions	Access and Read Definition
Modify objects	Administrator
Process objects	Access, Read Definition, and Process
Add or delete objects	Administrator
Set permissions	Administrator
Receive traces	Administrator

NOTE You can test security in your application by impersonating roles or specific users. Set the `Effective Roles` property in your connection string to a comma-delimited set of roles you want to impersonate, or set the `Effective Username` connection string property to the name of the user. Note that only server administrators can connect with these properties.

For example, you could use the following:

```
svr.Connect("location=localhost;" _ &
     "Initial Catalog=MyDatabase;Effective Roles=LimitedAccessRole")
```

Object Creation

To create mining models programmatically using AMO, you perform all the same steps you would perform if you were creating and managing the models in the user interface. That is, create a database, data source, data source view, mining structure, and mining model.

To create any object on the server, you generally perform the following steps:

1. Instantiate the object.
2. Set the object `Name` and `ID` properties.
3. Set the object-specific properties.
4. Add the object to its parent container.
5. Call `Update` to the object or its parent.

For example, Listing 16-1 demonstrates how to connect to a local server and create a database.

```
// Connect to the Analysis Service server
Server svr = new Server();
svr.Connect("localhost");
CreateDatabase();


Database CreateDatabase()
{
     // Create a database and set the properties
     Database db = new Database();
     db.Name = "Chapter 16";
     db.ID = "Chapter 16";

     // Add the database to the server and commit
     svr.Databases.Add(db);
     db.Update();

     return db;
}
```

Listing 16-1 Database creation

NOTE For simplicity, the rest of the listings in this chapter that include AMO sample code assume that `svr` is a member variable containing a connected Analysis Services server object.

Creating Data Access Objects

After you have an Analysis Services database object, the next step is to create `Datasource` and `DatasourceView` (DSV) objects. The `Datasource` object is fairly trivial, consisting of little more than a connection string to your database. The DSV is a bit more complicated. The main element of the DSV is the *schema*, which is a standard `Dataset` object augmented with custom properties.

To load a schema into a DSV, you create data adapters for each of the tables you want to load and add their schemas into a data set. You then add any relationships necessary, and finally add the data set to a DSV, which is then added to the AMO database.

Listing 16-2 demonstrates this procedure by creating a `Datasource` object for the `MovieClick` data (which is included in the Chapter 16 sample database, available in the `Chapter16.zip` archive for this chapter at `www.wiley.com/go/data_mining_SQL_2008`) and a DSV that can be used to create mining models with a nested table needed for analysis of movie channels.

```
void CreateDataAccessObjects(Database db)
{
    // Create a relational data source
    // by specifying the name and the id
    RelationalDataSource ds = new RelationalDataSource("MovieClick",
      Utils.GetSyntacticallyValidID("MovieClick", typeof(Database)));
    ds.ConnectionString = "Provider=SQLNCLI10.1;Data Source=localhost;
       Integrated Security=SSPI;Initial Catalog=Chapter 16";
    db.DataSources.Add(ds);

    // Create connection to datasource to extract schema to a dataset
    DataSet dset = new DataSet();
   SqlConnection cn = new SqlConnection("Data Source=localhost; Initial
       Catalog=Chapter 16; Integrated Security=true");

    // Create data adapters from database tables and load schemas
    SqlDataAdapter daCustomers = new SqlDataAdapter(
                    "SELECT * FROM Customers", cn);
    daCustomers.FillSchema(dset, SchemaType.Mapped, "Customers");

    SqlDataAdapter daChannels = new SqlDataAdapter(
                    "SELECT * FROM Channels", cn);
    daChannels.FillSchema(dset, SchemaType.Mapped, "Channels");

    // Add relationship between Customers and Channels
    DataRelation drCustomerChannels = new DataRelation(
            "Customer_Channels",
```

Listing 16-2 Data access object creation

```
              dset.Tables["Customers"].Columns["SurveyTakenID"],
              dset.Tables["Channels"].Columns["SurveyTakenID"]);
        dset.Relations.Add(drCustomerChannels);

        // Create the DSV, ad the dataset and add to the database
        DataSourceView dsv = new DataSourceView("SimpleMovieClick",
                             "SimpleMovieClick");
        dsv.DataSourceID = "MovieClick";
        dsv.Schema = dset.Clone();
        db.DataSourceViews.Add(dsv);

        // Update the database to create the objects on the server
        db.Update(UpdateOptions.ExpandFull);
    }
```

Listing 16-2 (*continued*)

TIP The ID of a `MajorObject` (database, data source, mining model, and others) must respect a set of syntactic restrictions. For example, certain special characters such as `?` or `!` cannot appear in a valid object ID. The `Utils.GetSyntacticallyValidID` function included in AMO generates a syntactically valid identifier starting from a given name and should always be used in real applications to generate valid IDs. It is only for simplicity reasons that this function is not used in the other code samples in this chapter.

The DSV in Listing 16-2 contains the `Customers` table and the `Channels` table, but the models you want to build need more specific information than is present in the raw data — in particular, the generation that the customers belong to (such as Baby Boomer or GenX) and a list of the premium movie channels they watch. To accomplish this, you must modify the code to add a calculated column to the `Customers` table, and swap out the `Channels` table with a named query that returns only the limited set of channels you are interested in.

Listing 16-3 contains `CreateDataAccessObjects` modified with a named calculation and named query.

```
    void AddNewDataAccessObjects(Database db)
    {
        // Create connection to datasource cto extract schema to a dataset
        DataSet dset = new DataSet();
        SqlConnection cn = new SqlConnection("Data Source=localhost;
```

Listing 16-3 Creating calculated columns and named queries

```
            Initial Catalog=Chapter 16; Integrated Security=true");

      // Create the Customers data adapter with the calculated appended
      SqlDataAdapter daCustomers = new SqlDataAdapter(
              "SELECT *, " +
              "(CASE WHEN (Age < 30) THEN 'GenY' " +
              " WHEN (Age >= 30 AND Age < 40) THEN 'GenX' " +
              "ELSE 'Baby Boomer' END) AS Generation " +
              "FROM Customers", cn);
      daCustomers.FillSchema(dset, SchemaType.Mapped, "Customers");
      // Add Extended properties to the Generation column indicating to
      // Analysis Services that it is a calculated column
      DataColumn genColumn = dset.Tables["Customers"].Columns
        ["Generation"];
      genColumn.ExtendedProperties.Add("DbColumnName", "Generation");
      genColumn.ExtendedProperties.Add("Description",
        "Customer generation");
      genColumn.ExtendedProperties.Add("IsLogical", "true");
      genColumn.ExtendedProperties.Add("ComputedColumnExpression",
                          "CASE WHEN (Age < 30) THEN 'GenY' " +
                          "WHEN (Age >= 30 AND Age < 40) THEN 'GenX' " +
                          "ELSE 'Baby Boomer' END");

      // Create a 'Pay Channels' data adapter with a customer query
      // for our named query
      SqlDataAdapter daPayChannels = new SqlDataAdapter(
           "SELECT * FROM Channels " +
           "WHERE Channel IN ('Cinemax', 'Encore', 'HBO', 'Showtime', " +
           "'STARZ!', 'The Movie Channel')", cn);
      daPayChannels.FillSchema(dset, SchemaType.Mapped, "PayChannels");
      // Add Extended properties to the PayChannels table indicating to
      // Analysis Services that it is a named query
      DataTable pcTable = dset.Tables["PayChannels"];
      pcTable.ExtendedProperties.Add("IsLogical", "true");
      pcTable.ExtendedProperties.Add("Description",
                      "Channels requiring an additional fee");
      pcTable.ExtendedProperties.Add("TableType", "View");
      pcTable.ExtendedProperties.Add("QueryDefinition",
           "SELECT * FROM Channels " +
           "WHERE Channel IN ('Cinemax', 'Encore', 'HBO', 'Showtime', " +
           "'STARZ!', 'The Movie Channel')");

      // Add relationship between Customers and PayChannels
      DataRelation drCustomerPayChannels = new DataRelation(
            "CustomerPayChannels",
            dset.Tables["Customers"].Columns["SurveyTakenID"],
```

Listing 16-3 (*continued*)

```
                dset.Tables["PayChannels"].Columns["SurveyTakenID"]);
        dset.Relations.Add(drCustomerPayChannels);

        // Access the data source and the DSV created previously
        // by specifying the ID
       DataSourceView dsv = new DataSourceView("MovieClick", "MovieClick");
        dsv.DataSourceID = "MovieClick";
        dsv.Schema = dset.Clone();
        db.DataSourceViews.Add(dsv);

        // Update the database to create the objects on the server
        db.Update(UpdateOptions.ExpandFull);
    }
```

Listing 16-3 (*continued*)

Creating the Mining Structure

The next step in the data mining program is to create the mining structure that describes the domain of the problem in terms the data mining engine understands. You must create `MiningStructureColumn`s and specify their data types, content types, and data bindings to their source columns in the DSV. Listing 16-4 contains the code to create a mining structure that will allow you to analyze the relationships between generation and premium channels.

```
    MiningStructure CreateMiningStructure(Database db)
    {
        // Initialize a new mining structure
        MiningStructure ms = new MiningStructure(
                    "PayChannelAnalysis", "PayChannelAnalysis");
        ms.Source = new DataSourceViewBinding("MovieClick");

        // Create the columns of the mining structure
        // setting the type, content and data binding

        // User Id column
        ScalarMiningStructureColumn UserID = new
              ScalarMiningStructureColumn("UserId", "UserId");
        UserID.Type = MiningStructureColumnTypes.Long;
        UserID.Content = MiningStructureColumnContents.Key;
        UserID.IsKey = true;
        // Add data binding to the column
        UserID.KeyColumns.Add("Customers", "SurveyTakenID",
              System.Data.OleDb.OleDbType.Integer);
```

Listing 16-4 Creating the mining structure

```
        // Add the column to the mining structure
        ms.Columns.Add(UserID);

        // Generation column
        ScalarMiningStructureColumn Generation = new
              ScalarMiningStructureColumn("Generation", "Generation");
        Generation.Type = MiningStructureColumnTypes.Text;
        Generation.Content = MiningStructureColumnContents.Discrete;
        // Add data binding to the column
        Generation.KeyColumns.Add("Customers", "Generation",
              System.Data.OleDb.OleDbType.WChar);
        // Add the column to the mining structure
        ms.Columns.Add(Generation);

        // Add Nested table by creating a table column and adding
        // a key column to the nested table
        TableMiningStructureColumn PayChannels = new
              TableMiningStructureColumn("PayChannels", "PayChannels");
        PayChannels.ForeignKeyColumns.Add("PayChannels", "SurveyTakenID",
              System.Data.OleDb.OleDbType.Integer);

        ScalarMiningStructureColumn Channel = new
              ScalarMiningStructureColumn("Channel", "Channel");
        Channel.Type = MiningStructureColumnTypes.Text;
        Channel.Content = MiningStructureColumnContents.Key;
        Channel.IsKey = true;
        // Add data binding to the column
        Channel.KeyColumns.Add("PayChannels", "Channel",
              System.Data.OleDb.OleDbType.WChar);
        PayChannels.Columns.Add(Channel);
        ms.Columns.Add(PayChannels);

        // Add the mining structure to the database
        db.MiningStructures.Add(ms);
        ms.Update();

        return ms;
    }
```

Listing 16-4 (*continued*)

NOTE You may wonder why you specify that the column content is `Key` and also have to set the `IsKey` property to `True`. This is because of the extensibility in the content types defined in the OLE DB for Data Mining specification. Currently, Analysis Services supports three types of keys: `Key`, `Key Time`, and `Key Sequence`. Having a separate `IsKey` property allows you to take advantage of this extensibility in the future.

Creating the Mining Models

Finally, you are at the point where you can create the models you want to use to analyze your customers. In addition to a collection of columns, a structure contains a collection of models. For each model, you add the columns you want from the structure and set their usage to `Key`, `Predict`, or `PredictOnly`. Columns without a specified usage are assumed to be `Input`, so you do not need to explicitly set them. For columns that you want the algorithm to ignore, you simply do not add them to the model.

Listing 16-5 demonstrates how to create two models inside the structure you previously built. A parameterized cluster model is created, and then a tree model is built from a copy of that model.

```
void CreateModels(MiningStructure ms)
{
    MiningModel ClusterModel;
    MiningModel TreeModel;
    MiningModelColumn mmc;

    // Create the Cluster model and set the algorithm
    // and parameters
    ClusterModel = ms.CreateMiningModel(true,
          "Premium Generation Clusters");
    ClusterModel.Algorithm = "Microsoft_Clustering";
    ClusterModel.AlgorithmParameters.Add("CLUSTER_COUNT", 0);

    // The CreateMiningModel method adds
    // all the structure columns to the collection


    // Copy the Cluster model and change the necessary properties
    TreeModel = ClusterModel.Clone();
    TreeModel.Name = "Generation Trees";
    TreeModel.ID = "Generation Trees";
    TreeModel.Algorithm = "Microsoft_Decision_Trees";
    TreeModel.AlgorithmParameters.Clear();
    TreeModel.Columns["Generation"].Usage = "Predict";
    TreeModel.Columns["PayChannels"].Usage = "Predict";

    // Add an aliased copy of the PayChannels table to the trees model
    mmc = TreeModel.Columns.Add("PayChannels_Hbo_Encore");
    mmc.SourceColumnID = "PayChannels";
    mmc = mmc.Columns.Add("Channel");
    mmc.SourceColumnID = "Channel";
    mmc.Usage = "Key";

    // Now set a filter on the PayChannels_Hbo_Encore table and use it
```

Listing 16-5 Adding mining models to the structure

```
        // as input to predict other channels
        TreeModel.Columns["PayChannels_Hbo_Encore"].Filter =
              "Channel='HBO' OR Channel='Encore'";

        // Set a complementary filter on the payChannels predictable
        // nested table
        TreeModel.Columns["PayChannels"].Filter =
              "Channel<>'HBO' AND Channel<>'Encore'";

        ms.MiningModels.Add(TreeModel);

        // Submit the models to the server
        ClusterModel.Update();
        TreeModel.Update();
    }
```

Listing 16-5 (*continued*)

Processing Mining Models

The code for processing an object is trivial, consisting only of the `Process` method called with the appropriate options. In the example program, you could process an individual model, the mining structure, or the entire database as you choose. However, because processing can be a rather lengthy task, it would be nice to receive progress messages from the server for the duration. Luckily, the AMO contains a `Trace` object to handle this type of server interaction. Listing 16-6 demonstrates setting up a progress trace for a processing operation.

```
    void ProcessDatabase(Database db)
    {
        Trace t;
        TraceEvent e;
        // create the trace object to trace progress reports
        // and add the column containing the progress description
        t = svr.Traces.Add();
        e = t.Events.Add(TraceEventClass.ProgressReportCurrent);
        e.Columns.Add(TraceColumn.TextData);
        t.Update();

        // Add the handler for the trace event
        t.OnEvent += new TraceEventHandler(ProgressReportHandler);
        try
        {
            // start the trace, process of the database, then stop it
            t.Start();
```

Listing 16-6 Processing the database with progress reports

```
            db.Process(ProcessType.ProcessFull);
            t.Stop();
        }
        catch (System.Exception /*ex*/)
        {
        }

    }

    void ProgressReportHandler(object sender, TraceEventArgs e)
    {
        Console.WriteLine(e[TraceColumn.TextData]);
    }
```

Listing 16-6 (*continued*)

DETERMINING SERVER CAPABILITIES

When you're creating models on the server, it is useful to understand exactly what kinds of models you can create. Besides the built-in algorithms, there may be plug-in algorithms installed as well. Additionally, each algorithm supports a variety of parameters whose default values may vary depending on the server configuration, for example between the Standard and Enterprise editions of SQL Server.

The `MINING_SERVICES` and `MINING_PARAMETERS` schema rowsets exposed by Analysis Services contain descriptions of the available algorithms and their capabilities. You can use any client command API to access these schemas, or even better, you can use the object model provided in ADOMD.NET to iterate quickly through the server's data mining capabilities. The following code demonstrates how to iterate through the mining services and their respective parameters:

```
public void DiscoverServices()
{
    AdomdConnection connection = new AdomdConnection(
          "Data Source=localhost");
    connection.Open();
    foreach( MiningService ms in connection.MiningServices)
    {
        Console.WriteLine("Service: " + ms.Name);
        foreach( MiningServiceParameter mp in
            ms.AvailableParameters)
        {
            Console.WriteLine("  Parameter: " + mp.Name +
                "  Default: " + mp.DefaultValue);
        }
    }
    connection.Close();
}
```

Deploying Mining Models

After creating your models, you may find that you need to move them around to different servers. For example, you may need to move them from an analytical server to a production server for embedding into line-of-business applications, or maybe simply to share a model with a colleague who cannot physically access your servers.

Analysis Services provides a robust backup-and-restore API in AMO. However, these APIs are geared more toward OLAP objects than toward data mining objects. The APIs contain many options that are unnecessary for data mining and operate solely at the database level, which is generally too coarse for most data mining operations.

Because of the mismatch in the functionality provided and the functionality required in AMO, the deployment of data mining objects is handled through DMX via a command API. Using the DMX `EXPORT` and `IMPORT` commands, you can select the single model that performs best out of the forest of candidate models you created and deploy it alone, rather than deploying the entire database.

Listing 16-7 demonstrates how you can use ADOMD.NET to transfer individual models from your current server to your production server.

```
public void TransferModel()
{
    // Create connections to the source and destination server.
    AdomdConnection cnSource = new AdomdCommand(
     "Data Source=localhost; Initial Catalog=Chapter 16");
    AdomdConnection cnDest = new AdomdCommand(
     "Data Source=ProductionServer; Initial Catalog=Chapter 16");
    try
    {
        // Export the model to a share on the destination server.
        AdomdCommand cmdExport = new AdomdCommand();
        cmdExport.Connection = cnSource;
        cmdExport.CommandText =
     "EXPORT MINING MODEL GenerationTree " +
      "TO '\\\\ProdutionServer\\Transfer\\GenerationTree.abk' " +
      "WITH PASSWORD= 'MyPassword'";
       cnSource.Open();
        cmdExport.ExecuteNonQuery();

        // Import the model into the current database on the
        // destination server.
        AdomdCommand cmdImport = new AdomdCommand();
        cmdImport.Connection = cnDest;
        cmdImport.CommandText = "IMPORT FROM " +
             " 'c:\\Transfer\\GenerationTree.abk' " +
```

Listing 16-7 Exporting and importing mining models

```
                  " WITH PASSWORD= 'MyPassword' ";
            cnDest.Open();
             cmdImport.ExecuteNonQuery();
        }
        catch(Exception /*ex*/)
        {
        }
        cnSource.Close();
        cnDest.Close();
    }
```

Listing 16-7 (*continued*)

In this example, you simply move one model between servers. The `EXPORT` command is flexible enough to export multiple models or entire mining structures as well. If you need to reprocess the models on the destination server, you can append `INCLUDE DEPENDENCIES` to the `EXPORT` command, and the necessary `Datasource` and DSV objects will be included in the export package.

NOTE Because OLAP objects do not support object-level importing and exporting, you cannot use the `EXPORT` command to export OLAP mining models.

Setting Mining Permissions

After the models are built, processed, and deployed, you must assign permissions so that they can be accessed by client applications. Permissions in Analysis Services are managed by the coordination of two objects: a `Role` object (which belongs to the database and contains a list of members) and a `Permission` object belonging to the protected object (which refers to a role and specifies the access permissions of that role). Listing 16-8 demonstrates the creation of a role and assigning permissions.

```
    void SetModelPermissions(Database db, MiningModel mm)
    {
        // Create a new role and add members
        Role r = new Role("ModelReader", "ModelReader");

        r.Members.Add(new RoleMember("redmond\\jamiemac"));
        r.Members.Add(new RoleMember("redmond\\zhaotang"));
        r.Members.Add(new RoleMember("redmond\\bogdanc"));

        // Add the role to the database and update
```

Listing 16-8 Assigning mining model permissions

```
        db.Roles.Add(r);
        r.Update();


        // Create a permission object referring the role
        MiningModelPermission mmp = new MiningModelPermission();
        mmp.Name = "ModelReader";
        mmp.ID = "ModelReader";
        mmp.RoleID = "ModelReader";

        // Assign access rights to the permission
        mmp.Read = ReadAccess.Allowed;
        mmp.AllowBrowsing = true;
        mmp.AllowDrillThrough = true;
        mmp.ReadDefinition = ReadDefinitionAccess.Allowed;


        // Add permission to the model and update
        mm.MiningModelPermissions.Add(mmp);
        mmp.Update();
    }
```

Listing 16-8 (*continued*)

Browsing and Querying Mining Models

Creating and deploying models is only the beginning. The real fun starts when you take the power of the learned knowledge of your models and embed that directly into your applications. You can recommend products, manage inventory, forecast revenue, validate data, and perform countless other tasks limited only by your data and your imagination.

Predicting with ADOMD.NET

Let's start with an example of a basic prediction query using ADOMD.NET. Listing 16-9 demonstrates a typical example of query execution. Readers familiar with ADO.NET will notice that the only differences between the APIs thus far are the names of the data access classes. In fact, it is equally possible to use the ADO.NET classes to perform simple queries against Analysis Services. However, ADOMD.NET is optimized to work with the Analysis Services server and allows you to take advantage of additional Analysis Services features.

```
public void SimplePredictionQuery()
{
    AdomdConnection connection = new AdomdConnection();
    connection.ConnectionString =
          "Data Source=localhost; Initial Catalog=Chapter 16";
    connection.Open();

    AdomdCommand cmd = connection.CreateCommand();
    cmd.CommandText =
        "SELECT Predict(Generation) FROM [Generation Trees] " +
        "NATURAL PREDICTION JOIN " +
        "( SELECT  " +
        "    (SELECT 'Cinemax' AS Channel UNION " +
        "     SELECT 'Showtime' AS Channel) AS PayChannels " +
        ") AS T ";

    // execute the command and display the prediction result
    AdomdDataReader reader = cmd.ExecuteReader();
    if (reader.Read())
    {
        string predictedGeneration = reader.GetValue(0).ToString();
        Console.WriteLine(predictedGeneration);
    }
    reader.Close();
    connection.Close();
}
```

Listing 16-9 Executing a simple singleton prediction query

For simplicity, the rest of the code samples in this chapter assume that the containing class has a `connection` member variable of type `AdomdConnection`, which holds an initialized connection.

Use `ExecuteReader` when executing queries that return multiple columns or rows, as shown in Listing 16-10. This performs the same prediction as in Listing 16-9, but it returns the flattened result of `PredictHistogram` so that you can see the likelihood of all possible prediction results.

```
public void MultipleRowQuery()
{
    AdomdCommand cmd = connection.CreateCommand();

    cmd.CommandText =
        "SELECT FLATTENED PredictHistogram(Generation) " +
       "FROM [Generation Trees] " +
        "NATURAL PREDICTION JOIN " +
        "( SELECT  " +
```

Listing 16-10 Iterating a multiple-row result

```
            "    (SELECT 'Cinemax' AS Channel UNION " +
            "     SELECT 'Showtime' AS Channel) AS PayChannels " +
            ") AS T ";
        AdomdDataReader reader = cmd.ExecuteReader();
        try
        {
            for (int i = 0; i < reader.FieldCount; i++)
            {
                Console.Write(reader.GetName(i) + "\t");
            }
            Console.WriteLine();

            while (reader.Read())
            {
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    object value = reader.GetValue(i);
                    string strValue = (value == null) ?
                        string.Empty : value.ToString();
                    Console.Write(strValue + "\t");
                }
                Console.WriteLine();
            }
        }
        finally
        {
            reader.Close();
        }
    }
```

Listing 16-10 (*continued*)

NOTE If your application reuses an `AdomdConnection` object for multiple queries, then you should ensure that any `AdomdDataReader` object is closed. A connection cannot execute a command while a reader is opened on that connection, and an exception thrown while reading data (and not handled properly) may lead to a leaked open reader that makes the connection unusable. The `finally` block at the end of the iteration in Listing 16-10 guarantees that the `connection` member variable is still available and in a valid state, even if reading from the reader throws an exception.

In the preceding example, you flatten the results of a nested table query for ease of iteration. In some situations, however, flattening the results is not practical. For example, this is not practical if you have a query that returns multiple nested tables, or even nested tables inside nested tables. Listing 16-11 demonstrates how to iterate the results of the previous example with the `FLATTENED` keyword removed.

```
while (reader.Read())
{
    for( int i = 0; i < reader.FieldCount; i++)
    {
        // Check for nested table columns
        if (reader.GetFieldType(i) == typeof(AdomdDataReader))
        {
            // fetch the nested data reader
            AdomdDataReader nestedReader = reader.GetDataReader(i);
            while (nestedReader.Read())
            {
                for (int j = 0; j < nestedReader.FieldCount; j++)
                {
                    object value = nestedReader.GetValue(j);
                    string strValue = (value == null) ?
                    string.Empty : value.ToString();
                    Console.Write(strValue);
                }
                Console.WriteLine();
            }
            // close the nested reader
            nestedReader.Close();
        }
    }
}
```

Listing 16-11 Iterating the Attribute column of the nested PredictHistogram result

Everything that you've done thus far could also have been done with ADO.NET (albeit, less efficiently). Next, you'll expand your application's functionality by using a parameterized query to change the prediction input. ADO.NET does not support named parameters for providers other than the SQL Server relational engine. To use named parameters in your data mining query, you must use ADOMD.NET, as demonstrated in Listing 16-12.

```
cmd.CommandText =
    "SELECT Predict(Generation) FROM [Generation Trees] " +
    "NATURAL PREDICTION JOIN " +
    "( SELECT  " +
    "    (SELECT @Channel1 AS Channel UNION " +
    "     SELECT @Channel2 AS Channel) AS PayChannels " +
    ") AS T ";

AdomdParameter p1 = new AdomdParameter();
p1.ParameterName = "Channel1";
```

Listing 16-12 Data mining query with named parameters

```
p1.Value = "Cinemax";
cmd.Parameters.Add(p1);

AdomdParameter p2 = new AdomdParameter();
p2.ParameterName = "Channel2";
p2.Value = "Showtime";
cmd.Parameters.Add(p2);
```

Listing 16-12 (*continued*)

Listing 16-12 assumes that you allow and require only two channels to perform the prediction. Obviously, this is not always the case. ADOMD.NET allows you use a parameter to pass an entire table as the input data source. This enables you to easily perform predictions using data that is on the client or otherwise unavailable to the server. Multiple table parameters may be shaped together to represent nested tables. Listing 16-13 demonstrates how you can use shaped table parameters as prediction input.

```
    AdomdCommand cmd = connection.CreateCommand();

    cmd.CommandText =
        "SELECT Predict(Generation) FROM [Generation Trees] " +
        "NATURAL PREDICTION JOIN " +
        "SHAPE { @CaseTable } " +
        "   APPEND( { @NestedTable } RELATE CustID TO CustID) " +
        "   AS PayChannels " +
        "AS T ";

    DataTable caseTable = new DataTable();
    caseTable.Columns.Add("CustID", typeof(int));
    caseTable.Rows.Add(0);

    DataTable nestedTable = new DataTable();
    nestedTable.Columns.Add("CustID", typeof(int));
    nestedTable.Columns.Add("Channel", typeof(string));
    nestedTable.Rows.Add(0, "Cinemax");
    nestedTable.Rows.Add(0, "Showtime");


    AdomdParameter p1 = new AdomdParameter();
    p1.ParameterName = "CaseTable";
    p1.Value = caseTable;
    cmd.Parameters.Add(p1);

    AdomdParameter p2 = new AdomdParameter();
```

Listing 16-13 Data mining query with table parameters

```
        p2.ParameterName = "NestedTable";
        p2.Value = nestedTable;
        cmd.Parameters.Add(p2);

        // execute the command and display the prediction result
        AdomdDataReader reader = cmd.ExecuteReader();
        if (reader.Read())
        {
            string predictedGeneration = reader.GetValue(0).ToString();
            Console.WriteLine(predictedGeneration);
        }
        reader.Close();
```

Listing 16-13 (*continued*)

NOTE The `SHAPE` statement builds a hierarchical rowset out of two flat rowsets, based on a parent-child relationship between the two rowsets. In Listing 16-13, the parent-child relationship links the `CustID` column in the top rowset (the `CaseTable` parameter) to the `CustID` column in the child rowset (the `NestedTable` parameter). A value of `0` is used in both rowsets to link `Cinemax` and `Showtime` together as nested table rows for a single top-level data row.

More on Table-Valued Parameters in ADOMD.NET

Listing 16-13 introduced table-valued parameters (in the form of `DataTable` objects) and used them for predictions. Table-valued parameters (also called *rowset parameters* in Books Online and various Analysis Service materials) are a very powerful feature of SQL Server Data Mining, because they allow data mining on any kind of application data.

Table-valued parameters may be used in prediction queries (such as in Listing 16-13), as well as in training mining structures and models (in conjunction with the DMX `INSERT INTO` statement). This allows applications to build mining models on-the-fly, without the need to stage the data in a relational database first. This feature is extensively used in the Table Analysis Tools for Excel 2007 add-in discussed in Chapter 2.

Note that a table-valued parameter need not be a .NET `DataTable` object. In .NET, a table-valued parameter may also be an implementation of the `IDataReader` interface, commonly implemented by various data access tools (such as ADO.NET). This allows Analysis Services to perform data mining on relational data that is not accessible because of network constraints.

Consider the scenario shown in Figure 16-2. An application has access to a local database and also to an Analysis Services HTTP endpoint, which resides outside of the local network. In this configuration, it is not possible to define binding on the Analysis Services server (bindings pointing to the local database). Therefore, the standard AMO method of building mining models cannot be used.

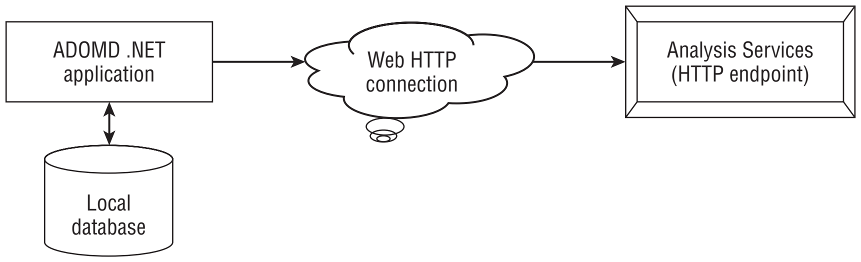

Figure 16-2 Data mining query with table parameters

However, the application may use table-valued parameters to upload training data from the local database to the Analysis Services instance. Listing 16-14 shows the code required to do this.

```
// Prepare one connection for each of the queries
SqlConnection relationalCnTop = new SqlConnection(
      "Data Source=localhost; Initial Catalog=Chapter 16;" +
      "Integrated Security=true");
SqlConnection relationalCnNested = new SqlConnection(
        "Data Source=localhost; Initial Catalog=Chapter 16;" +
      "Integrated Security=true");
// Open the local relational connections
relationalCnTop.Open();
relationalCnNested.Open();
SqlCommand cmdTop = relationalCnTop.CreateCommand();
SqlCommand cmdNested = relationalCnNested.CreateCommand();

// Prepare the relational queries
cmdTop.CommandText =
    "SELECT [SurveyTakenID], "+
    "   (CASE WHEN (Age < 30) THEN 'GenY' "+
    "     WHEN (Age >= 30 AND Age < 40) THEN 'GenX' "+
    "    ELSE 'Baby Boomer' END) AS Generation " +
    "FROM Customers " +
    "ORDER BY [SurveyTakenID]";


cmdNested.CommandText =
"SELECT * FROM Channels " +
"   WHERE Channel IN ('Cinemax', 'Encore', 'HBO',  " +
"                       'Showtime', 'STARZ!',
                            'The Movie Channel') " +
```

Listing 16-14 Uploading training data with table-valued parameters

```
        " ORDER BY [SurveyTakenID] ";

        // Create an Adomd command for Analysis Services
        AdomdCommand cmd = connection.CreateCommand();

        // Unprocess the mining structure, to make sure INSERT INTO
               will work
        cmd.CommandText = "DELETE FROM PayChannelAnalysis";
        cmd.ExecuteNonQuery();

        // Now prepare the INSERT INTO command
        cmd.CommandText = "INSERT INTO PayChannelAnalysis( " +
        "UserId, Generation, PayChannels( SKIP, Channel) ) " +
        "SHAPE { @CaseTable } " +
        "   APPEND( { @NestedTable } RELATE SurveyTakenID TO
                 SurveyTakenID) " +
        "AS PayChannels";


        // Add table valued parameters to the Adomd command
        // The parameters are added as IDataReader objects
        AdomdParameter p1 = new AdomdParameter();
        p1.ParameterName = "CaseTable";
        p1.Value = (IDataReader)cmdTop.ExecuteReader();
        cmd.Parameters.Add(p1);

        AdomdParameter p2 = new AdomdParameter();
        p2.ParameterName = "NestedTable";
        p2.Value = (IDataReader)cmdNested.ExecuteReader();
        cmd.Parameters.Add(p2);

        // Execute the training query
        cmd.ExecuteNonQuery();

        // close the relational connections
        relationalCnTop.Close();
        relationalCnNested.Close();
```

Listing 16-14 (*continued*)

Listing 16-14 uses two relational connections (to a local relational database) to execute two queries: one for the top-level data and one for the nested table data. The query results are passed to the `AdomdCommand` object as `IDataReader` objects, a forward-only interface that does not cache all the rows in memory (as in the case of a data table). Therefore, large volumes of data may be uploaded to Analysis Services without overloading the memory of the client application.

Browsing Models

ADOMD.NET provides a rich object model for browsing the content and metadata of the mining objects on a server that are otherwise accessible only through schema rowsets. Figure 16-3 shows the major data mining objects of ADOMD.NET.

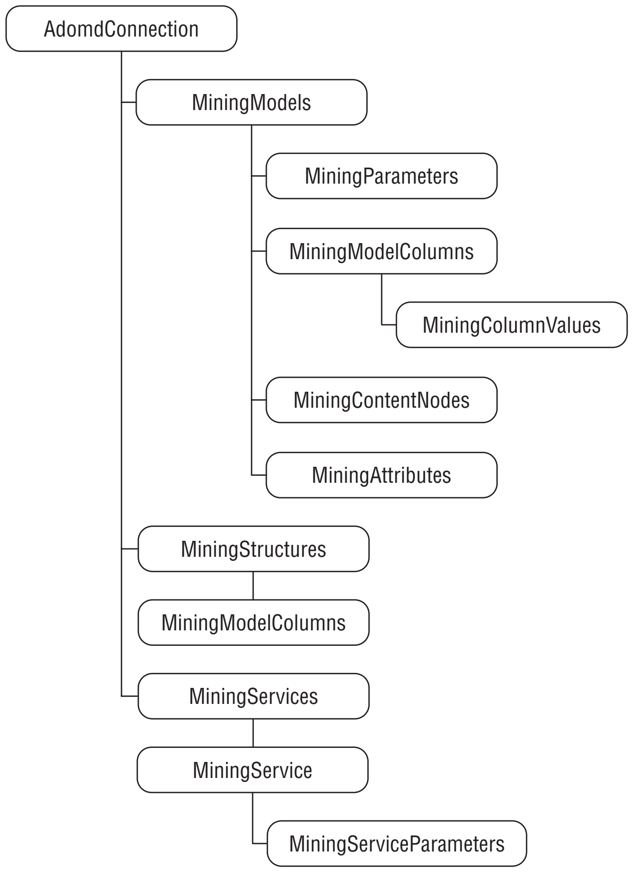

Figure 16-3 Data mining object hierarchy in ADOMD.NET

As you can see from the object model, you can simply connect to the server and iterate over any of the data mining objects without having to resort to schema queries. A nice benefit to application developers is that if a connected user does not have access to a particular object, that object will simply not appear in its collection (as if it didn't exist).

The most interesting thing you gain by using the ADOMD.NET object model is the capability to iterate mining model content in a natural, hierarchical manner using objects instead of trying to unravel the flat schema rowset form. Using this object model makes it easy to write complex programs to explore or display the content to your users. For example, an interesting problem for the Microsoft Decision Trees algorithm is to find all of the trees that contain a split on a given attribute.

Listing 16-15 demonstrates how you can use the content object model to explore trees and find splits on a specified attribute. First, you identify all child nodes of the root that represents trees, and then recursively check the children of the trees to see whether their marginal rule contains the requested attribute. By looking at the node type rather than at the algorithm used, this function will work against any model that contains trees, whether it uses the Microsoft Decision Trees algorithm, the Microsoft Time Series algorithm, or any third-party tree-based algorithm.

```
// Identify all the attributes that split on a specified attribute
public void FindSplits(string ModelID, string AttributeName)
{
    // Access the model and throw an exception if not found
    // The error text will be propagated to the client
    MiningModel model = connection.MiningModels[ModelID];
    if (model == null)
    {
        throw new System.Exception("Model not found");
    }

    // Look for the attribute in all model trees
    foreach (MiningContentNode node in model.Content[0].Children)
    {
        if (node.Type == MiningNodeType.Tree)
        {
            FindSplits(node, AttributeName);
        }
    }
}

// Recursively search for the attribute among content nodes
// return when children are exhausted or attribute is found
private void FindSplits(MiningContentNode node, string AttributeName)
{
    // Check for the attribute in the MarginalRule
    // and add row to the table if found
    if (node.MarginalRule.Contains(AttributeName))
    {
        Console.WriteLine(node.Attribute.Name);
        return;
    }
```

Listing 16-15 Exploring content using ADOMD.NET

```
        // recurse over child nodes
        foreach (MiningContentNode childNode in node.Children)
        {
            FindSplits(childNode, AttributeName);
        }

    }
```

Listing 16-15 (*continued*)

You can also use the `PredictNodeId` function to find the reason for a prediction. For example, you can use the following query to retrieve the ID of the node used to generate the prediction:

```
SELECT Predict(Generation), PredictNodeId(Generation) ...
```

You can then feed the result of this query into a function such as the one shown in Listing 16-16.

```
    public string GetPredictionReason(string ModelID, string NodeID)
    {
        // return the node description
        if (connection.MiningModels[ModelID] == null)
            throw new Exception("Model not found");
        MiningContentNode node =
         connection.MiningModels[ModelID].GetNodeFromUniqueName(NodeID);
        if( node == null )
            throw new Exception("Node not found");
        return node.Description;
    }
```

Listing 16-16 Using ADOMD.NET to find the reason behind a prediction

Stored Procedures

ADOMD.NET provides an excellent object model for accessing server objects and browsing content. However, there are some major drawbacks.

For the `FindSplits` method in Listing 16-15, you must bring the entire content from the server to the client to determine the list. A model with 1,000 trees and 1,000 nodes per tree would require the marshaling of more than 1,000,000 rows, even if only a handful of trees referenced the desired attribute. Also, in the `GetPredictionReason` function, even though you can access the desired node directly using `GetNodeFromUniqueName`, you are still causing a round trip to the server on each call. Performing this operation in batch mode is not recommended.

There is a solution to these problems. Analysis Services, starting with SQL Server 2005 (and continuing in SQL Server 2008) supports stored procedures that can be written in any managed language such as C#, VB.NET, or managed C++. The object model is almost identical to the object model of ADOMD.NET, making conversion between the two models simple. The clear advantage of Server ADOMD.NET is that all of the content is available on the server, and you can return only the information you need to the server. You can call user-defined functions (UDFs) by themselves, using the `CALL` syntax or as part of a DMX query. For example, the following query calls a stored procedure directly and simply returns the result:

```
CALL Chapter16SP.TreeHelper.FindSplits('Generation Trees','HBO')
```

The following query calls a stored procedure for every row returned from the prediction query:

```
SELECT Predict(Generation),
 [Chapter16SP].TreeHelper.GetPredictionReason(PredictNodeId(Generation))
 ...
```

In this case, the query will return the prediction result, plus the explanation of the result for every row.

CALLING VBA AND EXCEL FUNCTIONS AS STORED PROCEDURES

If you have Microsoft Office installed on the same machine as your Analysis Services server, you can leverage the functions of Visual Basic for Applications (VBA) and Excel as stored procedures inside your DMX queries.

For example, you can convert the prediction output to lowercase like this:

```
SELECT LCase(Predict([Generation])) FROM [Generation Trees]
PREDICTION JOIN ....
```

If a function exists in both Excel and VBA, you must prefix the function name with the name of the function. For example, to get the base 10 log of a prediction from Excel, and the natural log of the prediction from VBA, you would issue a query like this:

```
SELECT Excel!Log(Predict(Sales)), VBA!Log(Predict(Sales))
From MyModel ....
```

If an Excel function or VBA function also exists in MDX or DMX, or contains a $ character, you must escape the function name with square brackets (`[ ]`). For example, to format a prediction as currency (such as $20.56), you would issue a query like this:

```
SELECT [Format](Predict(Sales), '$d.dd') FROM MyModel ....
```

The supported VBA and Excel functions are listed in Appendix B.

Writing Stored Procedures

After you reference the required assembly (`Microsoft.AnalysisServices.AdomdServer`), you have access to a global object called `Context`. This object is similar to the ADOMD.NET connection object in that it contains all collections of major objects (such as `MiningModels`) that you can access in your stored procedure. The server-side `Context` object exposes a property, `CurrentMiningModel`,with no correspondent in the client side object model. This property provides the model that is the subject of the query and can be used in user-defined functions, as you will see in one of the procedures in Listing 16-17.

Stored procedures can take any simple type as a parameter and can return simple types or even a `DataTable` or `DataSet` in response. A client using `CALL` to call a stored procedure that returns a simple type will not receive a value, although the stored procedure will be executed. A client calling a stored procedure inside a prediction query that returns a `DataTable` or `DataSet` will receive a nested table containing the returned rows.

SENDING COMPLEX TYPES TO STORED PROCEDURES

If you need to send complex types (such as structures or arrays) to a stored procedure, you can serialize them using the `System.Xml.Serialization.XmlSerializer` object on the client and send them as a string. On the server side, deserialize the structure or array and call an overloaded function using the complex types you are interested in. For example, you may have a function that requires an array of the following type:

```
public struct MyType
{
     public int a;
     public string b;
}
```

You could write the following function to serialize the array into an XML string and send that string as a parameter to a stored procedure:

```
public string SerializeMyTypeArray(MyType[] arr)
{
    System.Xml.Serialization.XmlSerializer s =
      new System.Xml.Serialization.XmlSerializer(arr.GetType());
    System.IO.StringWriter sw = new System.IO.StringWriter();
    s.Serialize(sw, arr);
    string str = sw.ToString();
}
```

On the server side, you would need to duplicate the type definition and write a stub function to deserialize the array and call the real function, as follows:

```
public DataTable MySProc(string xmlString)
{
```

(continued)

SENDING COMPLEX TYPES TO STORED PROCEDURES *(continued)*

```
        MyType[] arr = null;
        System.Xml.Serialization.XmlSerializer s =
            new System.Xml.Serialization.XmlSerializer(typeof
                (MyType[]));
        System.IO.StringReader sr = new
              System.IO.StringReader(xmlString);
        arr = s.Deserialize(sr);
        return MySProc(arr);
    }
    protected DataTable MySProc(MyType[] arr)
    {
    ...
    }
```

This strategy will allow you to pass complex types and will prepare you for future versions that may allow complex types to be naturally passed.

Depending on the complexity of your data, you may want to use built-in .NET data types that support serialization (such as a `DataSet` object).

Stored Procedures and Prepare Invocations

When writing a procedure to be executed on the server, you need to know when you are being called to return a result versus when you are being called simply to gather schema information during a prepare call. Additionally, you need to indicate that your procedure is safe to call during a prepare operation and that calling it won't have any undesirable side effects. You wouldn't want to create the same object twice, for example.

The `Context` object contains an `ExecuteForPrepare` property that you can check before performing any time-consuming operations in your procedure. If you are returning a `DataTable` or `DataSet`, you should fully define the objects and return them empty of data so the client will know the schema. In general, you should not raise errors during preparation, especially for missing objects, because the prepare call could be used during a batch query, and the objects may exist by the time the procedure is called to return a result.

To indicate that your procedure does not have any unwanted side effects, you must add the custom attribute `SafeToPrepare`.

A Stored Procedure Example

Listing 16-17 demonstrates a stored procedure written in C#. The methods are the same as in Listing 16-15 and 16-16, but they are modified to operate on the server, take into account the presence of the `Context` object, and properly handle situations where the procedure is called during a prepare operation.

```
 [SafeToPrepare(true)]
public DataTable FindSplits(string ModelID, string AttributeName)
{
    // Create the result table and add a column for
    // the attribute
    DataTable tblResult = new DataTable();
    tblResult.Columns.Add("Attribute", typeof(string));

    // If this is a Prepare statement, return the empty table
    // for schema information
    if (Context.ExecuteForPrepare)
        return tblResult;

    // Access the model and throw an exception if not found
    // The error text will be propagated to the client
    MiningModel model = Context.MiningModels[ModelID];
    if (model == null)
    {
        throw new System.Exception("Model not found");
    }

    // Look for the attribute in all model trees
    if (model.Content.Count > 0)
    {
        foreach (MiningContentNode  node in model.Content[0].Children)
        {
            if (node.Type == MiningNodeType.Tree)
            {
                FindSplits(node, AttributeName, ref tblResult);
            }
        }
    }
    // return the table containing the full result
    return tblResult;
}

private bool FindSplits(MiningContentNode node, string AttributeName,
          ref DataTable tblResult)
{
    // Check for the attribute in the MarginalRule
    // and add row to the table if found
    if (node.MarginalRule.Contains(AttributeName))
    {
        string[] row = new string[]{node.Attribute.Name};
        tblResult.Rows.Add(row);
        return true;
```

Listing 16-17 Data mining stored procedure

```
        }

        // recurse over child nodes
        foreach (MiningContentNode childNode in node.Children)
        {
            if( FindSplits(childNode, AttributeName, ref tblResult) )
            {
                return true;
            }
        }

        return false;
    }

    [SafeToPrepare(true)]
    public string GetPredictionReason(string NodeID)
    {
        // return immediately if executing for prepare
        if (Context.ExecuteForPrepare)
            return string.Empty;

        // return the node description
       return     Context.CurrentMiningModel.GetNodeFromUniqueName(NodeID)
             .Description;
    }
```

Listing 16-17 (*continued*)

SUMMARY OF SIGNIFICANT DIFFERENCES BETWEEN ADOMD.NET AND SERVER ADOMD.NET

Following is a summary of significant differences between ADOMD.NET and Server ADOMD.NET:

- **Many of the functions of the client-side `Connection` object are covered by the `Context` object.**
- **There is no `Console.WriteLine` or any direct output on the server side. Results must be collected in a `DataTable` before being returned to the caller.**
- **Server-side code should handle statement preparation (the `SafeToPrepare` attribute).**
- **The server-side object model uses the concept of a `CurrentMiningModel`, which identifies the model targeted by the current query.**

Executing Queries inside Stored Procedures

A common use of a stored procedure is to encapsulate a query for easy reuse. For example, if your application needs to predict `Generation`, but you need the flexibility to change the model being used or add more business logic, you could write a procedure that executes the query and redeploy the procedure as necessary without changing the application layer.

Server ADOMD.NET allows you to execute DMX queries using the same objects that you would use with ADOMD.NET. The only exception is that you do not have to specify a connection, because you are already connected. You can copy the results from the query into a `DataTable`, or you can simply return the `DataReader` returned by `ExecuteReader`.

Listing 16-18 demonstrates the query from Listing 16-9 implemented as a UDF.

```
using Microsoft.AnalysisServices.AdomdServer;
using System.Data;
...
[SafeToPrepare(true)]
public IDataReader PredictGeneration()
{
   // Create a new AdomdCommand object
   // Note how it does NOT use a connection. The command is implicitly
   // connected to the database on which the stored procedure is invoked
   AdomdCommand cmd = new AdomdCommand();

   // Use an empty table to create a reader with the same shape
   // as the result for Prepare
    if (Context.ExecuteForPrepare)
    {
        DataTable tbl = new DataTable();
        tbl.Columns.Add("Generation", typeof(string));
        return tbl.CreateDataReader();
    }

   // Initialize the command with a query
    cmd.CommandText =
        "SELECT Predict(Generation) FROM [Generation Trees] " +
        "NATURAL PREDICTION JOIN "+
        "( SELECT  " +
        "    (SELECT 'Cinemax' AS Channel UNION " +
        "     SELECT 'Showtime' AS Channel) AS PayChannels " +
        ") AS T ";
    return cmd.ExecuteReader();
}
```

Listing 16-18 Executing a DMX query inside a stored procedure

In this example, if you want to change the model that's performing the prediction, you could just change the query inside the stored procedure, without having to change queries embedded inside your application. Of course, you can also parameterize your query as previously demonstrated in Listing 16-12.

NOTE Stored procedures cannot be used to implement security in Analysis Services. The security context of the current user determines the access to the objects inside the Analysis Services server. That is, any user who calls a procedure that queries a mining model but who does not have Read permission on that model will receive a permission error. Similarly, a user who calls the `GetPredictionReason` UDF from Listing 16-17 but who does not have Browse permission on the model will also receive a permission error.

Returning Data Sets from Stored Procedures

Server-side procedures used as UDFs inside a query may return tabular- or scalar-type data. Procedures invoked with `CALL` typically need to return tabular content (that is, a `DataTable` or an `IDataReader` implementation).

Server-side stored procedures also have the ability to return `DataSet` objects (including multiple data tables and possibly various relationships defined between tables). `DataSet` results cannot be displayed in tabular result viewers (such as SQL Server Management Studio, which treats them as strings), but can be used programmatically. Listing 16-19 shows a server-side stored procedure that returns a `DataSet` object, and Listing 16-20 shows the client-side ADOMD.NET code that consumes the `DataSet` results.

```
public DataSet GetSimpleDependencyNet(string ModelID)
{
    // define the first table in the data set
    // It has two columns, AttributeID(int) and AttributeName (string)
    DataTable tblAttributes = new DataTable();
    tblAttributes.Columns.Add("AttributeID", typeof(int));
    tblAttributes.Columns.Add("AttributeName", typeof(string));

    // define the second table in the data set. Each row indicates a
    // dependency in the net (an arrow).
    // It has two numeric columns, each indicating the source attribute
    // and the second indicating the target
    DataTable tblRelationships = new DataTable();
    tblRelationships.Columns.Add("From", typeof(int));
```

Listing 16-19 Stored procedure that computes a dependency network and returns it as a DataSet object

```
        tblRelationships.Columns.Add("To", typeof(int));


        // Access the model and throw an exception if not found
        // The error text will be propagated to the client
        MiningModel model = Context.MiningModels[ModelID];
        if (model == null)
        {
            throw new System.Exception("Model not found");
        }

        // Build a hash table which contains all attributes and their IDs
        Dictionary<string, int> dictAttributes = new
                Dictionary<string, int>();
        foreach (MiningAttribute att in model.Attributes)
        {
            dictAttributes.Add(att.Name, att.AttributeID);
        }

        // Now traverse all the predictable attributes, call the FindSplits
        // procedure and insert the results in the table
        foreach (MiningAttribute att in model.Attributes)
        {
            // Add one row to tblAttributes for each model attribute
            object[] attributeRow = new object[2];
            attributeRow[0] = att.AttributeID;
            attributeRow[1] = att.ShortName;
            tblAttributes.Rows.Add(attributeRow);


            DataTable tblSplitsOnThisAttribute =
                   FindSplits(ModelID, att.ShortName);
            // add one row to tblRelationships for each attribute that
            // depends on the current one
            object[] relationshipsRow = new object[2];
            relationshipsRow[0] = att.AttributeID;
            foreach (DataRow row in tblSplitsOnThisAttribute.Rows)
            {
                // FindSplits returns a 1-column table containing
                // attribute names
                string dependentAttribute = row[0] as string;
                relationshipsRow[1] = dictAttributes[dependentAttribute];
                tblRelationships.Rows.Add(relationshipsRow);
            }
        }
```

Listing 16-19 (*continued*)

```
        // Add both tables to a data set object
        DataSet ds = new DataSet();
        ds.Tables.Add(tblAttributes);
        ds.Tables.Add(tblRelationships);

        // define a relationship inside the dataset
        ds.Relations.Add("FromAttributeName",
                   tblAttributes.Columns["AttributeID"],
                   tblRelationships.Columns["From"]);
        ds.Relations.Add("ToAttributeName",
                   tblAttributes.Columns["AttributeID"],
                   tblRelationships.Columns["To"]);

        // return the data set
        return ds;
    }
```

Listing 16-19 (*continued*)

```
        AdomdCommand cmd = connection.CreateCommand();
        DataSet resultDS = null;
        // Invoke the DataSet returning stored proc
        cmd.CommandText =
           "CALL [Chapter16SP].GetSimpleDependencyNet('Generation Trees')";

        // The first value returned by the query is the data set
        AdomdDataReader rdr = cmd.ExecuteReader();
        if (rdr.Read())
        {
           object obj = rdr.GetValue(0);
           if (obj is DataSet)
               resultDS = (DataSet)obj;
        }
        rdr.Close();

        DataTable tblAttributes = resultDS.Tables[0];
        DataTable tblRelationships = resultDS.Tables[1];

        DataRelation fromAttName = resultDS.Relations["FromAttributeName"];
        DataRelation toAttName = resultDS.Relations["ToAttributeName"];

        foreach (DataRow relRow in tblRelationships.Rows)
        {
                string attFrom =
```

Listing 16-20 Using ADOMD.NET to consume the DataSet results from a stored procedure

```
                relRow.GetParentRow(fromAttName)["AttributeName"] as string;
            string attTo =
                relRow.GetParentRow(toAttName)["AttributeName"] as string;
            Console.WriteLine(string.Format("{0} -> {1}", attFrom, attTo));
           }
```

Listing 16-20 (*continued*)

The stored procedure presented in Listing 16-5 traverses all attributes in a mining model and invokes the previously defined `FindSplits` procedure to detect all attributes that depend on the current attribute (effectively building a dependency network of all attributes in the mining model). The dependency network is serialized in a data set that contains two tables: a table of the attributes' IDs and names, and a table of edges (directed dependencies between attributes).

Deploying and Debugging Stored Procedure Assemblies

After you have compiled and built your stored procedure, you must deploy the procedure to your Analysis Server so that you can call it from DMX. To add a .NET assembly to your Analysis Services project, right-click the Assemblies folder in Solution Explorer and select New Assembly Reference.

When deploying an assembly, you will need to select some security-related options, such as `Permissions` and `Impersonation` information. The defaults are functional for most scenarios. However, the discussion in this section provides some details on these options.

The `Permissions` property specifies the code access permissions that are granted to the assembly when it's loaded by Analysis Services. The recommended (and default) value is `Safe`. Table 16-4 shows the possible values and their implications. As a developer of stored procedures, you may need less-restrictive permissions (such as network access, if your stored procedure attempts to connect to an external data source). As a system administrator, you will need to decide what permissions would be best to balance your system's safety and functionality.

You can use the `Impersonation` property of an assembly when you require the stored procedure code to run under certain credentials. However, you cannot use `Impersonation` to control access to the internal object model. Object model access is always performed under the credentials of the current user. However, if your stored procedure connects to a remote database, you may need to require your assembly to impersonate the current user.

When you deploy your project, your assembly is encoded and sent to Analysis Server, where it is available for use in the project database. When you need to update your assembly, you can simply redeploy it. If you are using a live project, the assembly is immediately deployed on the server. To update an assembly in a live project, delete the assembly and add it back to the project.

Table 16-4 Permissions for Stored Procedure Assemblies

VALUE	DETAILS
`Safe`	This is the most restrictive and safest permission set. The code executed by the assembly cannot access external system resources (files, the network, or the registry).
`External access`	The assembly code may access files, the network, and the registry.
`Unrestricted`	The assembly code is completely unrestricted and may call any managed or unmanaged code.

If you have a general-purpose assembly that you want to access across all databases on the server, you can use SQL Server Management Studio to deploy it at the server level. In Object Explorer, right-click the Assemblies collection of the server, select Add Assembly, and then select the assembly you want to add.

Debugging assemblies is best done when you're running the server and client on the same machine. You can use a development license of SQL Server for this purpose. To debug the assembly in Visual Studio, select Attach to Process from the Debug menu. Select the executable `msmdsrv.exe` from the list, and ensure that the dialog box displays Common Language Runtime as the Attach To option. After you have followed these steps, you will be able to set breakpoints in your stored procedures.

Summary

In this chapter, you learned about the variety of APIs that you can use to access the functionality of Analysis Services programmatically. Although many APIs are supported, the two most important APIs are AMO and ADOMD.NET. You can use AMO to programmatically create, process, and manage your mining models, structure, and servers. ADOMD.NET is the general client API for browsing and prediction queries.

Using these APIs, you can create intelligent applications of your own. The logic of your application can involve dynamically creating mining models to solve user-defined problems. It can apply the predictive power of the data mining algorithms or examine the learned content of the mining models to provide new insights and new abilities to your users. You can also leverage your server in your application by writing UDFs that have access to all of the server resources through a .NET programming model.

The sample code for this chapter is available at `www.wiley.com/go/data_mining_SQL_2008`. It consists of three projects that exemplify AMO (the `AMO-Management` project), ADOMD.NET (the ADOMD-`BrowseAndQuery` project), and Server ADOMD.NET (the `Chapter16SP` project). The ADOMD.NET sample application depends on the mining models and structure created by the AMO application, as well as some stored procedures included in the Server ADOMD.NET sample project. Therefore, the sample projects should be built and deployed in this order: AMO, Server ADOMD.NET, and then ADOMD.NET.

In Chapter 17, you will learn how to extend the set of features that are built into Analysis Services, such as custom data mining algorithms and viewers.