Application Programmer's Manual

MetaCube ROLAP Option

for Informix Dynamic Server

Version 4.1
December 1998
Part No. 000-5226

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Table of Contents

Chapter 7 The Folders Class of Objects

Chapter 8 The Queries Class of Objects and Related Collections

Introduction

In This Introduction

This introduction provides an overview of the information in this manual and describes the conventions this manual uses.

About This Manual

This manual contains information to assist you in using the Informix-MetaCube OLE automation programming interface to create custom applications.

Organization of This Manual

The chapters in this manual describe the classes of objects that together form the MetaCube programming interface. Each chapter contains information about one or more related classes of objects, including detailed descriptions of the properties and methods available for each class of object. This manual is designed as a reference manual. Programmers who are new to MetaCube should read this Introduction, Chapter 1, "Object-Oriented Programming and the MetaCube API," and Chapter 2, "Getting Started: MetaCube Tutorial." Refer to all other chapters in this manual as necessary. The manual consists of the following chapters:

- Chapter 1, "Object-Oriented Programming and the MetaCube API," provides a quick introduction to object-oriented programming, OLE Automation, Visual Basic for Applications, and the MetaCube programming interface.
- Chapter 2, "Getting Started: MetaCube Tutorial," provides an extensive tutorial that explains how to develop a sample application.

- Chapter 3, "The Metabase Class of Objects," describes the **Metabase** class of objects. Objects in this class represent virtual multidimensional databases.
- Chapter 4, "The Dimensions Class of Objects and Related Collections," describes the **Dimensions** class of objects and its related classes of objects. The **Dimensions** class and its related collections allow you to develop procedures that create, edit, or access MetaCube's metadata.
- Chapter 5, "Extensions," describes the programming interface used to incorporate extensions compiled in C++ into MetaCube. This chapter also describes the functions created as extensions that are distributed with MetaCube.
- Chapter 6, "The FactTables Class of Objects and Related Collections," describes the **FactTables** class of objects and its related classes of objects. The **FactTables** class and its related collections allow you to develop procedures that create, edit, or access MetaCube's metadata.
- Chapter 7, "The Folders Class of Objects," describes how to save query and filter definitions in folders.
- Chapter 8, "The Queries Class of Objects and Related Collections," describes the properties and methods available for these two classes and their related collections.
- Chapter 9, "The Schemas Class of Objects and Its Collections," describes the **Schemas** class of objects and its hierarchy of Table and Column collections.
- Chapter 10, "Users and DSS Systems," describes how user and DSSSystem objects can be manipulated to support the security features of MetaCube Secure Warehouse.
- Chapter 11, "The SystemMessages Class of Objects," describes the **SystemMessages** class of objects, which allow you to distribute messages to users within an DSS System.
- Chapter 12, "The ValueList,"explains how to return multiple values into a development environment.
- Chapter 13, "The Applications Class of Objects and Global Properties," provides a brief explanation of the **Applications** class of objects, the highest-level object class for any OLE software server.

- Chapter 14, "Scoping Rules," explains rules for identifying objects with the same name but belonging to different parents or different classes.

Types of Users

This manual is written for programmers who are developing custom MetaCube applications. You should be experienced with Object Linking and Embedding (OLE) and Visual Basic (VB). You should also be familiar with MetaCube Explorer.

Additional Documentation

Other printed manuals for the Informix-MetaCube product are:

- **MetaCube Explorer User's Guide**. This manual is written for people who are responsible for analyzing data about their company's business. It describes how to query the data warehouse in multidimensional terms to obtain meaningful reports that are the basis for timely business decisions.
- **MetaCube for Excel User's Guide.** This manual is written for people who use Microsoft's Excel spreadsheet for business analysis. After adding MetaCube for Excel to the Excel software, the Excel user can query a data warehouse in multidimensional terms to obtain spreadsheet or PivotTable reports.
- **Warehouse Manager's Guide**. This manual is written for the data warehouse administrator and describes how to specify internal information about the data warehouse so that the MetaCube components are able to access and graphically present the database for querying.
- **MetaCube SDK for Snap-Ins Programmer's Manual**. This manual is written for the C++ programmer who will write custom extensions for MetaCube Explorer and MetaCube for Excel. The MetaCube SDK for Snap-Ins product includes an Extension Wizard that generates skeletal C++ code, which can be modified to provide customized analysis functions.

- **MetaCube and MetaCube Agents Installation and Configuration Guide**. This manual describes how to install and configure the MetaCube software components both on the server and on PCs.

Readme Files

In addition to the printed manuals, README files are distributed with the Informix-MetaCube product. These files contain technical information, including last-minute changes to product capability or documentation. Please read these files. They contain important information.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Drive, Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

doc@informix.com

We appreciate your suggestions.

Chapter

1

Object-Oriented Programming and the MetaCube API

In This Chapter

This chapter discusses object-oriented programming, OLE-Automation, Visual Basic for Applications, and the MetaCube programming interface, introducing six major classes of MetaCube objects.

OLE, OLE Automation, and MetaCube

OLE (object linking and embedding) is an object technology developed by Microsoft Corporation to enable application developers to build and integrate component software. OLE is based on an advanced underlying object architecture called the OLE Component Object Model (COM), which is the basis of Microsoft's strategy to evolve the whole family of Windows operating systems into object-based operating systems and application environments. COM, the model on which OLE is built, is an underlying system software object model that allows complete interoperability between software components developed by different vendors, even when those components are programmed in different languages.

In its original incarnation, OLE supported compound documents (for example, documents that included a spreadsheet from a different application) or drag-and-drop. This manual primarily discusses *OLE Automation*, because this technology allows developers to access MetaCube's full range of functionality from their own development environments.

OLE Automation is unrelated to linking and embedding. OLE Automation is a mechanism that allows applications to expose their internal objects and the command sets that control and manipulate those objects to the operating system. Services can vary widely from application to application, but they are provided through a standard object interface.

For example, through OLE Automation, you can activate a spreadsheet application, populate and format specific cells, and then deploy the spreadsheet's charting function to create a graph. This is possible even for users without programming experience, or programmers unfamiliar with the underlying OLE model.

Corporate developers, system integrators, and power users can use traditional programming languages, development tools, and even productivity tools to access these capabilities. For example, a user could manipulate a spreadsheet using the standard macro language in his or her word processor.

This technique it allows corporate developers and system integrators to quickly assemble larger, customized business solutions using packaged, component software as building blocks.

One such building block is MetaCube. MetaCube is implemented as an OLE Automation server, so that all of its internal objects, such as queries and filters, are available for use by any application that supports OLE. Currently, all of the major development environments, including Visual Basic, C++, PowerBuilder, and SQLWindows, feature built-in support for OLE Automation. In addition, several of the major productivity tools, including Microsoft Word, Excel, and Access, support OLE Automation. Consequently, MetaCube can serve as the basis for high-performance, multidimensional access to your data warehouse, regardless of your development environment.

Introduction to Object-Oriented Programming

To leverage the MetaCube analysis engine, you must first understand object-oriented programming. Object-oriented programming encapsulates functionality within objects. Each object represents a set of properties; each property stores data defining the object. For example, a rectangle object might have properties such as its width, height, and position. These properties define the object.

To deploy an object, you must issue a *message* to that object, changing a property's value, or activating a *method*. The methods of an object define how that object accomplishes different tasks. Unlike procedural programming, in which properties are defined and methods executed independently, an object incorporates both properties and methods, which not only define an entity but also the operations that can be performed by, with, or on that entity.

For example, the rectangle object might include a method to move the rectangle. Rather than designing a separate procedure to re-position the rectangle, as you would in a procedural programming environment, you can issue a message to the rectangle object, which executes a standard method to move the rectangle. The procedure for moving a rectangle depends on the size and original position of the rectangle, as the rectangle essentially must be re-drawn. A rectangle object however, with properties of size and original position, may also include a method for moving itself, which you can activate by sending a message to the object.

Because an object's methods are self-contained, the object can be called from any development environment. Whereas the procedure for moving a rectangle differs from one language to another, the message issued to an object remains the same, regardless of the development environment. The movement of the rectangle object is a self-contained process.

Object Classes

Although we can discuss rectangles generally, and the properties that define rectangles, such as their position and shape, a program must define a particular rectangle, with a particular position and a particular shape. Each rectangle is a separate object, and all rectangle objects belong to the rectangle object class.

An object *class* represents a general type of object, and an *instance* of an object class is a particular object of that class's type. An instance of an object class is often simply referred to as an object. When you develop an object-oriented application, you create or edit an instance of an object class, investing the general properties of its class with particular values. An object class may have, for example, a property such as color, whereas an instance of that object class may assign a value to that property, such as red, green, or blue.

Aside from its open database connectivity (ODBC) interface, MetaCube consists of a library of object classes created in C++. To develop MetaCube applications, you do not create new object classes. A MetaCube application simply issues messages, in the form of OLE Automation calls, to this library of object classes, instantiating objects, assigning values to their properties, and invoking their methods.

The relationship between object classes and an instance of an object class (or simply, an object) is analogous to the relationship between the type of car one owns—say, Ford—and the actual car itself: *my* Ford with the dented fender and the broken headlight. Just as you can only drive an instance of the Ford class of cars, you can only program with an instance of an object class. The Ford class has properties, such as the speed at which it is driven, or the position of its rearview mirror, but those properties only acquire specific values when you drive an instance of a Ford, that is, an actual car, at a speed of thirty miles per hour, and a mirror tilt of twenty degrees.

Object Class Hierarchies and Collections

Object classes are organized hierarchically, with objects of the same class assembled in a *collection* belonging to a particular parent object. A collection of objects can partially define the parent object to which that collection belongs. For example, the Ford can own a collection of car door objects, each with different properties, such as their respective positions, and different methods, such as opening, closing, or rolling down a window. Together, this collection of door objects partially defines the Ford. Each door, in turn, can own a collection of knobs or controls, such as a lock or a handle, that partially define the door. The MetaCube analysis engine, like the example of the car, consists of a hierarchy of object classes. Each instance of most object classes own a collection containing instances of another object class. In fact, several collections of objects of different classes may belong to a single parent object.

A collection of objects is specified by identifying the parent of that collection, followed by the plural of the class name to which the objects in that collection belong:

```
MyFord.CarDoors
```

A dot separates the terms of this statement, in this case the name of the parent and the collection, respectively.

To determine how many objects exist within a collection, you can deploy the **Count** property, a general property of all collections that return the number as an integer value. For the collection of car doors, for example, the **Count** property may represent a value of 4, which we can display in a MsgBox object, a Visual Basic for Applications object for displaying values that is used throughout this guide:

```
MsgBox MyFord.CarDoors.Count
```

To determine the names of objects within a collection, you can deploy the **Names** property, a general property of all collections that have objects with a name property, such as Dimensions, Attributes, and DSSSystems. The **Names** property returns a ValueList containing the names of the objects in the collection:

```
MsgBox MyFord.CarDoors.Names
```

To perform different operations on items in any collection, you can invoke the methods summarized in Figure 1-1.

Figure 1-1
General Methods of a Collection

Method	Description/Example
Add	This method adds an instance of an object class to a collection. The arguments necessary, if any, depend on the properties of the added item. `Parent.Collection.Add Argument1, Argument2, Argument3...`
Item	This method identifies a particular instance or item within a collection by a sequentially generated index number or by the name assigned to the instance. Once you have identified the item, you deploy one of that item's methods or properties. `Parent.Collection.Item(Index Number or Item Name).Action`
MakeFirst	This method and the two that follow re-arrange the order of items within a collection. This method in particular arranges the specified item as the first item in the collection, with a new index number of 0. You must identify the item by name or by its original index number. `Parent.Collections.MakeFirst "This Item"`

(1 of 2)

Method	Description/Example
MakeLast	This method rearranges items within a collection such that the specified item is the last item in the collection. You can identify this item by name or by its original index number. `Parent.Collections.MakeLast "This Item"`
MakeNth	This method assigns the specified item to a different location within the collection, with a new index number. You must identify the item to be moved by name or by its original index number, followed by its new index number. `Parent.Collections.MakeNth "This Item", 2`
Remove	This method removes an item from a collection; if the item does not exist in an overlapping or overarching collection, this method deletes the item. You must identify the item to be removed by name or by index number. If you are removing an object from the Extensions collection, you must refer to the item by index number. `Parent.Collection.Remove "This Item"`
RemoveAll	This method removes all items from a collection. `Parent.Collection.RemoveAll`
Swap	This method exchanges the positions of two elements in a collection. Since this method is commutative, you can identify the two items to swap by name or by index number, in either order. `Parent.Collection.Swap 1, 3`

(2 of 2)

While MetaCube's object classes are organized hierarchically, this hierarchy does not involve *inheritance*, as supported by development environments such as PowerSoft's PowerBuilder. Inheritance allows developers to invest an object with the properties of another object, in essence cloning that object. The new object can then be further developed, with its properties constituting a superset of the old object's properties. The OLE standard does not support inheritance.

In a MetaCube hierarchy, each class of objects is defined by a unique set of properties, which do not overlap. The example of the car object illustrates the distinction: although a car object is in part defined by a collection of car door objects, the car and the car door have different sets of properties, with neither object encompassing the other object's properties as a subset of its own.

Figure 1-2 describes the major MetaCube object classes used to build and execute queries.

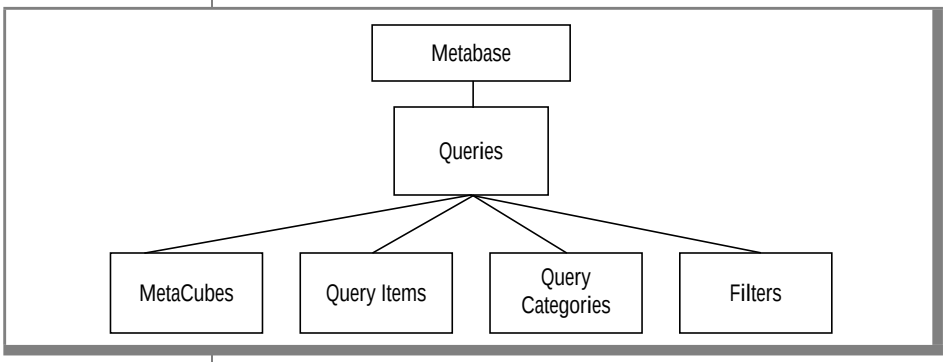


Figure 1-2 *Simplified Representation of the Hierarchy of MetaCube Object Classes*

Figure 1-3 provides a complete view of all MetaCube object classes and their hierarchical relationships.

Figure 1-3
Complete MetaCube Class Hierarchy

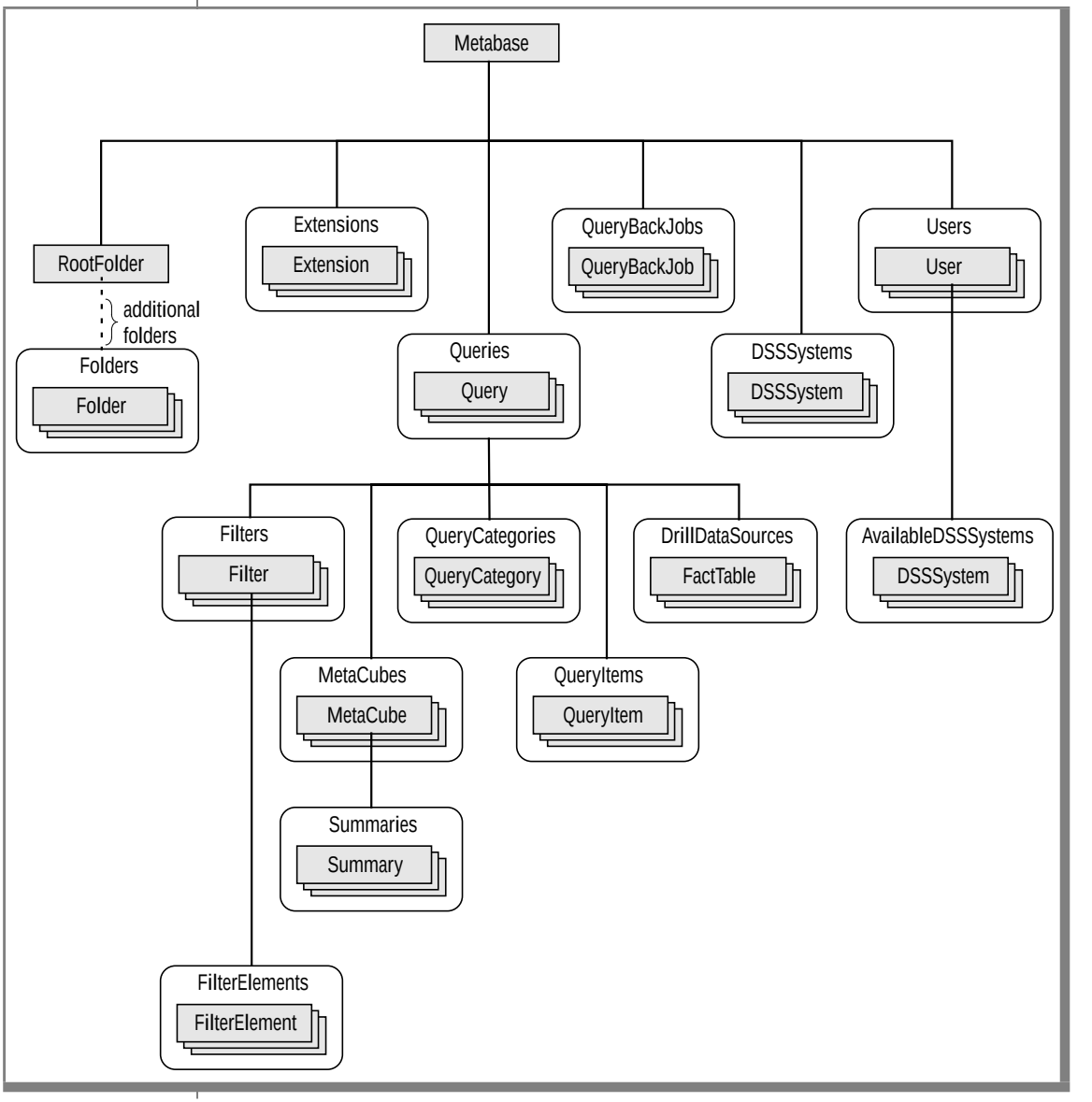
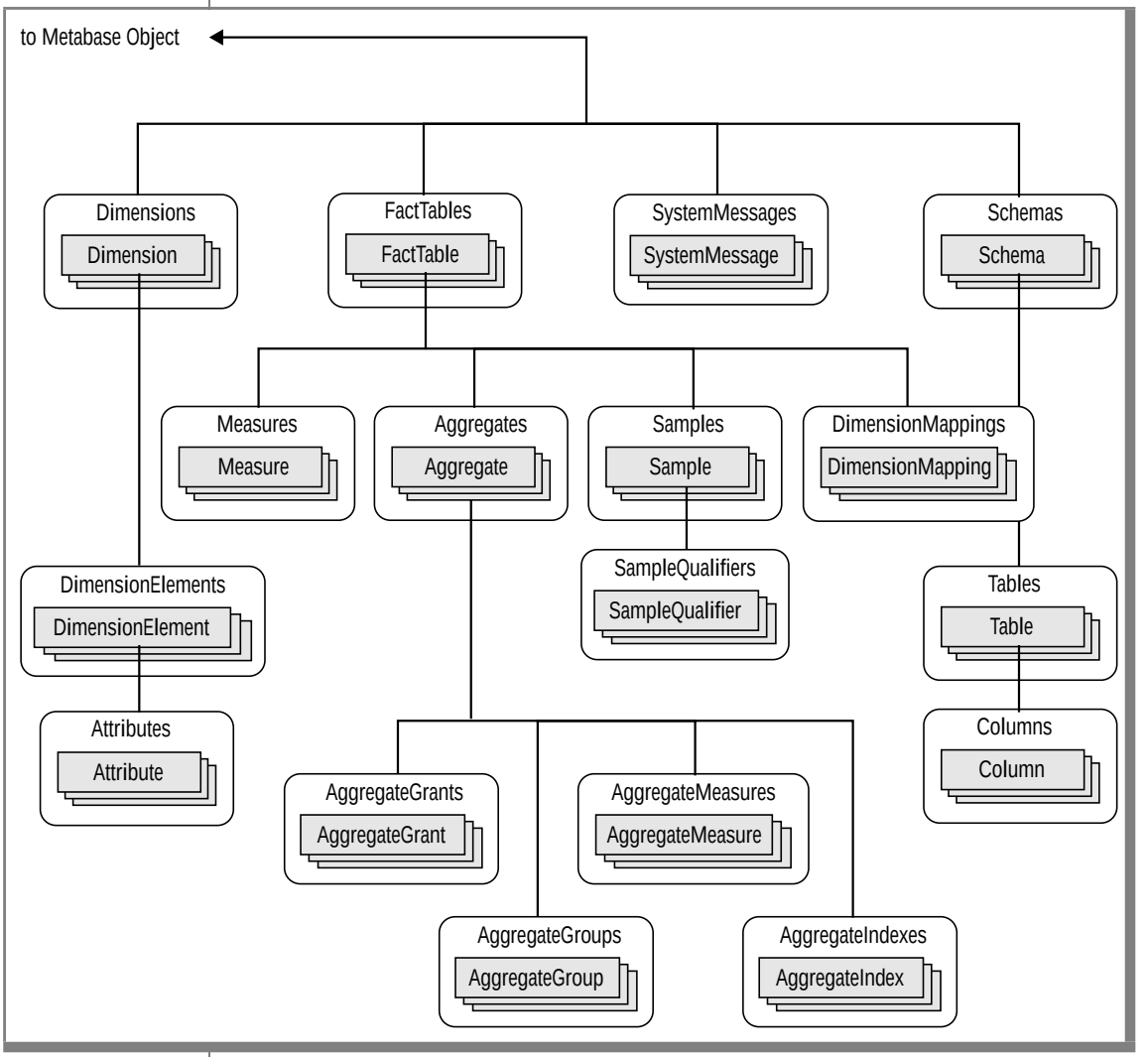


Figure 1-3 (continued)
Complete MetaCube Class Hierarchy (continued)



The **Metabase** class of objects stands at the top of MetaCube's hierarchy of object classes as the master object. Each instance of the **Metabase** class of objects represents a "virtual" multidimensional database (a "metabase"), which offers a multidimensional view of tables in a relational database without actually storing data in a multidimensional format. The features of each metabase depend not only on the set of relational tables to which it corresponds but also on the metadata description of those tables. A complete metadata description of relational tables comprises a Decision Support Software System, or DSS System. Multiple descriptions of identical tables may exist, either for security purposes or to accommodate different communities of users. Every MetaCube procedure begins by instantiating a Metabase object. A full discussion of the Metabase object begins on page 3-3.

If the MetaCube analysis engine has not already been launched, instantiating a Metabase object launches the MetaCube analysis engine. The engine remains running until the instantiation is released from memory, usually at the end of the procedure. Upon release of an instance of a Metabase object, the engine may close, depending on whether the engine was running prior to that object's instantiation.

Different development environments can recognize the **Metabase** class of objects as a MetaCube object class because MetaCube registers the **Metabase** object class in your operating system's OLE registry upon installation.

Every instantiation of a Metabase object implicitly creates collections of child objects, the principal of these being a collection of query objects. Although the collections are initially empty, this semantic distinction is important because once you have explicitly instantiated a Metabase object, instances of other objects can simply be added to their respective, pre-existing collections. Unlike the **Metabase** class of objects, these objects do not require an explicit function for instantiation.

Queries that retrieve data through the particular multidimensional structure represented by a Metabase object belong to the collection of queries owned by that object. Each query consists of:

- multidimensional attributes, which correspond to the "what, when, and where" components of the query.
- measures, which correspond to the "how much" component of the query.
- filters, which set conditions limiting the range of data retrieved for a query.

- reports or, more precisely, multidimensional representations of a data set, which determine how the data retrieved for a query will be displayed.

Consider a typical query that retrieves the number of units sold, by region and by brand, for the last two weeks and displays the result in a cross-tabular report. The measure of the query is **Units Sold**, the attributes are **Brand** and **Region**, the filter defines a range of two weeks, and the report is cross-tabular. These terms should be familiar to you from your experience with MetaCube Explorer and are documented more fully in the *MetaCube Explorer User's Guide*.

A different object class represents each component of the query's definition. If a query is defined by multiple components of the same type, multiple objects of the same type are instantiated and stored in that query's collection of objects of that type. A given query may be defined by one or more attributes, measures, filters, and reports, with objects of a given class comprising a collection belonging to the query object. Query components' object class names differ slightly from their common names, as identified above.

Figure 1-4 describes the nomenclature for these four classes of objects.

Figure 1-4
Primary Collections of a Query Object

Common Name	Object Name	Function
Attributes	QueryCategories	"What," "when," and "where" components of a query
Measures	QueryItems	"How much" component of a query
Filters	Filters	Defines range of data retrieved
Reports	MetaCubes	Defines format in which to display data

Declaring MetaCube Object Type Variables

Using MetaCube and Visual Basic or Visual Basic for Applications, you can declare MetaCube-specific object type variables and then safely create new instances of the MetaCube object classes. This approach to creating objects yields multiple advantages, including faster processing time, type safety (Visual Basic issues an error if it receives the wrong type of object), and function checking during the coding process. If your MetaCube system operates over a network using a middle tier, faster processing is particularly apparent.

To declare object type variables in MetaCube, you must add a reference to the MetaCube type library (**metacube.tlb**) in your project's type library. Then use the **Dim** and **Set** statements. The **Dim** statement declares a variable that refers to an object. No actual object is created, however, until you use the **Set** statement with a **New** keyword. The following example illustrates how **Dim** is used to formally declare the variable **MyMetabase** as **Metabase**, and then a **Set** statement with the **New** keyword is used to create an instance of the Metabase object. Use the **New** keyword to create instances of a Metabase object only. To create instances of all other MetaCube objects, use the **Add** method for each class of object.

```
Dim MyMetabase as Metabase
Set MyMetabase = New Metabase
```

You might want to use object references to variables rather than create an instance of the article itself. Because object references to variables are references to (rather than copies of) the object, any change in the object is reflected in all variables that refer to it. The following example illustrates how to assign an object reference to a variable for a Query object.

```
Dim MyQuery as MetaCubeLibrary.Query
Set MyQuery = MyMetabase.Queries.Add("first query")
```

In the example above, the **Dim** statement is used to fully qualify the object declaration. Because object types in different type libraries may have the same name, you should include **MetaCubeLibrary** in an object declaration to prevent ambiguity.

When you make a new object reference to a Query object, enclose the name of the query added in parentheses. When using the **Set** statement in Visual Basic or Visual Basic for Applications, you must remember to put parentheses around any arguments. Functions that return values to variables require parentheses around arguments, but procedures do not. Simply adding a query without assigning that query to a particular object reference executes a procedure, while assigning the query to an object reference performs a function.

Finally, to minimize the system resources consumed by object references, you should release all object references at the close of your application, particularly since running the application again would otherwise entail creating and setting object references still held in memory:

```
MyMetbase.Queries.Remove(MyQuery)
Set MyQuery = Nothing
Set MyMetabase = Nothing
```

By convention, the object references in this guide are identified by the prefix ***My*** followed by the class name of the object referenced. In practice, you can name your object variables however you like.

Compatibility

Object type variables are supported by Visual Basic 4.0, Visual Basic 5.0, and Excel 97. Excel 95 does not support the use of object type variables.

Object Variables Used in MetaCube API Exercises

The MetaCube API exercises presented in this manual do not incorporate the use of object type variables. Instead they declare object variables, a programming technique supported by Visual Basic 4.0, Visual Basic 5.0, Excel 95, and Excel 97. The following example shows how an object variable is created in this manual's exercises. In the example, an object variable, **MyMetabase**, is declared, and then an instance of an object of the **Metabase** class is stored in the object variable **MyMetabase**.

```
Dim MyMetabase as Object
Set MyMetabase = CreateObject("Metabase")
```

Visual Basic for Applications

All examples in this text have been developed in Visual Basic for Applications (VBA), the macro language for Microsoft Excel, version 5.0 and later releases. Because most readers, regardless of their preferred development environment, have installed Microsoft Excel on their PCs, all examples use VBA syntax. For Visual Basic (VB) users, VBA offers another advantage, insofar as it is quite similar to VB and actually anticipates many of the new features incorporated into Visual Basic 4.0. Perhaps the most important distinction between VB and VBA is that, rather than storing code in text files, as VB does, VBA stores code in Excel's file format, as part of a workbook. For Microsoft Office '97 users, Excel might present a slightly different interface, which more completely resembles the Visual Basic development environment. To insert a macro module in an Excel '97 workbook, see the Excel documentation: the actions described here apply only to Excel '95 and previous releases.

To developers familiar with PowerBuilder, SQLWindows, C++, or other OLE-automated development environments, VBA might seem to differ quite drastically from the language to which they are accustomed. Although the syntax for beginning procedures, passing arguments, declaring variables, and displaying data differs across development environments, the OLE Automation calls to the MetaCube analysis engine remain more or less the same. This manual emphasizes MetaCube objects, with a minimum of VBA-specific information.

To develop a MetaCube application from Microsoft Excel, launch the application and open a new workbook. A new workbook typically consists of six or more sheets, which you can access by clicking the tabs, labeled **Sheet1**, **Sheet2**, and so on, at the bottom of Excel's main window. Within each worksheet, you can deploy Excel's standard spreadsheet capabilities. You can also use a worksheet to display data retrieved by MetaCube.

To develop applications in VBA, you must insert a macro module, the site where you actually enter, edit, and view code. Applications developed in VBA must always be executed from a module, or by a keystroke, menu item, or toolbar item linked to a macro module. To insert a macro module, choose **Insert→Macro→Module**. See Figure 1-5.

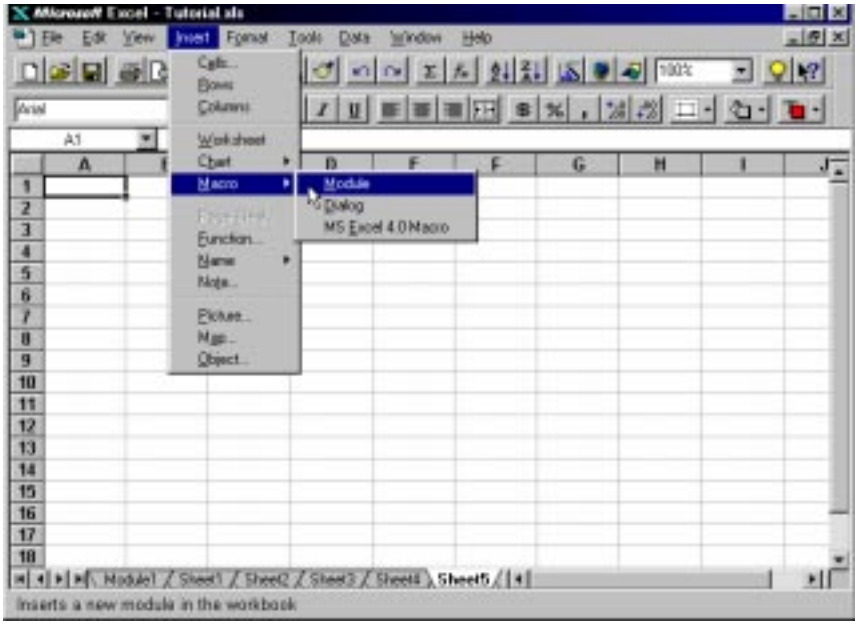


Figure 1-5
Inserting a Macro Module

A tab labeled **Module1** appears to the right of the worksheet tabs at the bottom of Excel's main window. To begin entering code, click this tab, which opens the macro module.

Renaming Worksheets and Macro Modules

You can also rename your macro module by right-clicking the module and selecting **Rename** from the popup menu. Rename **Module1** as **MetaCube Code**. Rename two other worksheets as **Define the Query**, and **Query Report**.

Once you have inserted a module, and given appropriate names to that module and several worksheets, you are ready to build a VBA application that makes OLE Automation calls to the MetaCube analysis engine.

Chapter

2

Getting Started: MetaCube Tutorial

In This Chapter

Using this chapter you can develop a sample application that employs many of the object classes, properties, methods, and programming techniques discussed throughout this manual.

MetaCube in Thirteen Lessons: An API Tutorial

Before thoroughly reviewing each MetaCube object class and its methods, properties, and collections, many developers and power users can gain a working understanding of MetaCube's application programming interface by developing a simple MetaCube query application.

Although MetaCube exposes objects that perform the work of Warehouse Manager, which maps the relational database as metadata, most custom-built applications revolve around building and executing multidimensional queries, referencing metadata created in Warehouse Manager. This discussion of MetaCube's objects begins with the task of building multidimensional queries. Multidimensional queries, which the *MetaCube Explorer User's Guide* discusses in greater detail, are defined by such natural business terms as time, product, and geography. MetaCube can be thought of as a virtual multidimensional database, translating multidimensional queries into ANSI-standard SQL.

This tutorial begins by hard-coding a simple query. It then generates a report for that query and subsequently adds filtering, calculations, pivoting, and sorting features. Ultimately, the procedure populates list boxes with the names of available attributes, measures, and saved filters, prompting the user to define the query in the terms listed. Each exercise adds to the body of code from the previous exercises, with the new material indicated in **bold**.

To begin using the tutorial

1. Load Microsoft Excel, version 5.0 or later.
2. Open the file **M3_API.XLS**, which installs in the MetaCube exercise directory.

   This workbook contains a set of named worksheets to which the exercises refer and a module for each of the exercises in this tutorial.
3. Create a new module to begin entering your own application code or simply refer to existing modules as you read through the tutorial.

The Option Explicit Statement and Comments

Line 1 of each exercise contains an optional statement, **Option Explicit**, which requires you to declare explicitly any variables introduced by a procedure before you use those variables. Enabling this option prevents VBA from mistaking a misspelled variable for an entirely new variable, and thus quickly identifies typographical errors.

The remainder of the line, marked by an apostrophe (‘), is a comment. VBA ignores comments. Most of the procedures cited in this text are commented for your benefit.

Connection Information in the MetaCube API Exercises

To perform any of the exercises in this manual, you must first connect to the database and the MetaCube analysis engine. Lines 16 through 26 in Exercise 1 provide an example of the necessary connection information. Because API Exercise 1 through Exercise 12 build on each other, connection information is repeated in each of those exercises. Exercise 14 through Exercise 24 do not follow an incremental pattern, however, and they do not all explicitly set connection information. Nonetheless, a connection must be established to execute any of those exercises, just as a connection is established in Exercise 1.

Default user connection properties can be set in MetaCube Secure Warehouse. You do not have to explicitly set **Metabase.ConnectString** or **Metabase.Name** (the DSS System name) if you have user properties defined in Secure Warehouse and you want those default connection properties to be used.

Launching MetaCube

As the logical representation of a multidimensional database, the Metabase object is MetaCube's master object, similar to Excel's Application object. Every MetaCube program begins by instantiating a Metabase object, storing that instantiation in an object variable. In VBA, the **CreateObject** function instantiates foreign objects, with the class name of the object passed as an argument to the function. Instantiating a Metabase object also requires an OLE Automation call to the MetaCube analysis engine, prompting the engine to launch. For the purpose of these exercises, you should launch the MetaCube analysis engine manually, from Windows, because your VBA application is otherwise forced to load MetaCube each time your application runs.

Exercise 1: The Metabase Object

Each Metabase object identifies a DSS System, which defines a particular multidimensional view of the relational database. Other properties assigned to the Metabase object typically correspond to other preferences and login information required in applications like MetaCube Explorer, such as the login, password, or host connection string. Once you have instantiated the Metabase object, and set certain Metabase properties, you can connect to the relational database, using the **Connect** method of the Metabase object.

```
1  Option Explicit 'All variables must be declared explicitly
2
3   'MetaCube API Exercise 1: The Metabase Object
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object
10
11  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
12      'Instantiate a Metabase object
13      Set MyMetabase = CreateObject("Metabase")
14
15      'Identify an ODBC Data Source
16      Let MyMetabase.ConnectString = "Metademo"
17
18      'Specify a set of metadata
19      Let MyMetabase.Name = "MetaCube Demo"
20
21      'Specify login to database
22      Let MyMetabase.Login = "metademo"
23      'Passes value to database on connection
24
25      'Specify database password
26      Let MyMetabase.Password = "Metademo"
27
28      'Log in to demonstration database, open DSS System
29      MyMetabase.Connect
30
31  End Sub
```

Explanation of Exercise 1

The procedure begins at line 5 with the **Sub** syntax, followed by the name of the procedure and any arguments passed to the procedure. Arguments are enclosed in parentheses. The empty set of parentheses signifies that no arguments have been passed to this procedure.

A single variable, **MyMetabase**, is declared at line 9 as an object type variable. A **dim** statement declares variables locally, within the scope of a procedure, whereas a **global** statement declares variables that are available to all modules in all workbooks. Global variables remain in memory until released, or until the workbook closes, and should be avoided when possible. Either syntax requires you to name the variable and allows you to specify its type, integer, string, object, and long. If you do not specify a variable's type, VBA assumes it is of the variant type. For more information, see "Declaring MetaCube Object Type Variables" on page 1-14.

Line 13 instantiates an object of the **Metabase** class. The class of the object is passed as an argument to the **CreateObject** function, and the instance of the class is stored in the object variable **MyMetabase**. For a full discussion of the relationship between an object class and an instance of that class, see "Object Classes" on page 1-5. All object variables must be "set" equal to a value, as shown here, while other types of variables allow the optional **let** syntax.

Lines 16, 19, 22, and 26 assign different properties to the Metabase object **MyMetabase**. The **ConnectString** property identifies an ODBC data source. The **Name** property identifies the multidimensional map of the relational database by which the MetaCube analysis engine configures itself to retrieve data from the RDBMS. Each mapping is a DSS System, created in Warehouse Manager, or through a similar application developed through the MetaCube programming interface. The value specified for this property, the **MetaCube Demo** DSS System, is a demonstration system referred to throughout MetaCube documentation.

The **Login** and **Password** properties allow you to specify, respectively, the name of the user/schema to which you are logging in on the relational database and the password for that user/schema. Except for several specialized server-side processes, MetaCube relies on the already rigorous role-based security of the RDBMS.

Once you have provided the password and login information, you can connect to the relational database using the **Connect** method, as shown in line 22. Values for any unspecified **Metabase** properties are read from the **metacube.ini** file. For a complete list of **Metabase** properties, see "Metabase Properties" on page 3-5.

The procedure ends on line 31 with the syntax **End Sub**, which automatically releases all locally declared variables. Once the instantiation of the Metabase object has been released, the MetaCube analysis engine disconnects from the relational database. If you did not launch the MetaCube analysis engine manually, the release of the Metabase object ultimately closes the engine. This procedure thus disconnects without attempting to build a query. Once you have successfully connected, you are ready to modify this procedure to define a query.

To execute this procedure, position the cursor between the first and the last line of code and press F5, or click the **Play** button on Excel's Visual Basic for Applications toolbar. To step through the procedure line-by-line, press F8.

Exercise 2: Defining A Query

A collection of query objects belongs to each Metabase object, and, in turn, collections of attributes (QueryCategories), measures (QueryItems), filters, and reports belong to each query object. See Figure 1-2 on page 1-9. By instantiating a Metabase object such as **MyMetabase**, you implicitly create the collections that belong to an object of this class. Rather than using the **CreateObject** function to instantiate a Query object, for example, you can simply add an instance of a query to an existing collection of Query objects. To add an object to a collection, identify the object being added to the collection, provide the name of the collection itself (typically the plural of an object class, such as Queries or Filters), and use the **Add** method, a general method for adding an item to any collection.

Each procedure in this set of exercises builds on the first, adding several lines for each new exercise. The added lines are set off in **boldface**, and explained after the exercise. This exercise defines a query and displays the SQL generated by the MetaCube analysis engine in a message box. This exercise does not issue that SQL to the database.

```
Option Explicit 'All variables must be declared explicitly

 'MetaCube API Exercise 2: Defining a Query

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         MyQuery As Object

     Const MyFirstAttribute = "Brand"
     Const MyMeasure = "Units Sold"

'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"

     'Specify a set of metadata
     Let MyMetabase.Name = "MetaCube Demo"

     'Specify login to database
     Let MyMetabase.Login = "metademo"
     'Passes value to database on connection

     'Specify database password
     Let MyMetabase.Password = "Metademo"

     'Log in to demonstration database, open DSS System
     MyMetabase.Connect

 'Define the Query
 Set MyQuery = MyMetabase.Queries.Add("A New Query")
 'Adds query to MyMetabase's collection of queries

 MyQuery.QueryCategories.Add MyFirstAttribute

 'Add measure to MyQuery's collection of measures
 MyQuery.QueryItems.Add MyMeasure
```

```
43
44   'Display Query Definition
45   MsgBox MyQuery.SQL
46   'SQL property shows SQL generated for query before execution
47
48   End Sub
```

Explanation of Exercise 2

This exercise declares a second object variable, **MyQuery**, in line 10, which subsequently stores the newly added instance of the **Queries** class of objects. You can declare variables in a list, marking each new variable by a comma and individually specifying the type of each.

In anticipation of the arguments necessary to define the query, you also declare two constants in lines 12 and 13, using the syntax **Const**, followed by the name of the constant, the operator =, and the value it stores. One constant specifies a measure defined in the metadata; the other specifies an attribute defined in the metadata. If, within a DSS System, a measure name is repeated in another fact table, or if an attribute name is repeated across dimensions, you must provide a more specific definition of that component, identifying the exact measure or attribute desired, as described in "Scoping Rules" on page 14-3. Although you could specify the attribute and measure names directly in the query definition, declaring constants in a single location allows you to easily change your query definition.

Lines 15 through 33 connect the MetaCube analysis engine to the relational database, as discussed in on page 2-6.

Lines 36 to 42 define a query by a single attribute and a single measure. Measures represent different types of numeric data associated with a transaction and correspond to the "how much" component of a query. Attributes represent different ways of grouping those measures and correspond to the "what, when, and where" components of a query. Every query that retrieves numeric data must be defined by at least one attribute and one measure. An attribute is incorporated into a procedure as a QueryCategory, and a measure is incorporated into a query as a QueryItem. This section of the procedure defines a query requesting sales, grouped by brand, where **Units Sold** is the measure and **Brand** is the attribute.

To define a query, you must instantiate an object of the **Queries** class. Each Query object belongs to a collection of Query objects, all of which descend from an instance of the **Metabase** class of objects. To instantiate a Query object, you must identify a particular Metabase object's query collection and deploy the general **Add** method, as shown on line 36.

The name of the query appears as an argument at the end of this command. Enclose the argument in parentheses because you are returning this instance of the **Queries** class of objects to an object variable, and functions require all arguments to be enclosed in parentheses. In MetaCube Explorer, new queries are given names such as **Untitled1** until a user saves the query under a different name. However, this application does not recognize your Query object by this name. This instance of the **Queries** class of objects is stored in the object variable **MyQuery**.

Lines 39 and 42 refer to the **MyQuery** object variable as the parent of several collections. As shown in Figure 1-2 on page 1-9, each Query object owns collections of QueryCategory, QueryItem, Filter, and MetaCube objects, which represent, respectively, attributes, measures, filters, and reports. To generate SQL for a standard query, you must include at least one QueryCategory in a Query object's collection of QueryCategories and one QueryItem in a Query object's collection of QueryItems.

Such collections are, of course, initially empty. Line 39 specifies **MyQuery**'s collection of QueryCategories, instantiating a QueryCategory object representing the **Brand** attribute within that collection. The name of the attribute appears as an argument at the end of this command. For the MetaCube analysis engine to understand this argument, you must create metadata for the attribute of that name, describing which tables and columns correspond to this attribute. The same is true of measures, filters, and other logical objects.

This instance of the QueryCategory object is not stored in an object variable, and any subsequent references to this object identifies this object by name or by index number as an item within the collection of QueryCategories owned by **MyQuery**. Because this command does not return a value, the constant representing the name of the attribute should not be enclosed in parentheses.

Line 42 identifies the measure included in this query definition, adding a QueryItem object to **MyQuery**'s collection of QueryItem objects. The name of the measure, as specified in your metadata, appears as an argument at the end of the command. A constant, **MyMeasure**, stores the name of the measure, **Units Sold**.

If you had mapped this measure to more than one fact table / data source, you would have included the name of the fact table when you specified the measure, for instance, **Sales Transactions.Units Sold**. If the measure corresponds to only one fact table, you do not need to include the name of the fact table.

As was the case when instantiating a QueryCategory object, no variable stores this instance of the **QueryItems** class of objects, and, consequently, you do not enclose this command's arguments in parentheses.

Once you have satisfied the minimum requirements for a query definition, we can view the SQL commands the MetaCube analysis engine would generate to retrieve the data requested by the query from the relational database. Line 45 invokes the SQL property of the **Queries** class of objects, which stores as a string the SQL generated for the query represented by **MyQuery**.

The Visual Basic for Applications' **MsgBox** function displays this string expression in a dialog box. As a query's definition becomes more complex, the message box may not be able to contain the entire set of SQL commands generated for the query, and the definition is cut off. A message box cannot contain more than 1,024 characters, although the exact limit depends in large part on the width of the characters.

Once your application displays the SQL generated by the query, the application terminates on line 48. The next exercise executes the SQL on the relational database.

Exercise 3: Executing the Query, Displaying the Results

In this exercise, you execute the query defined in the previous exercise using the **ToVBArray** method and display the results using Excel's **Range** object.

```
1  Option Explicit 'All variables must be declared explicitly
2
3   'MetaCube API Exercise 3: Executing the Query, Displaying the
   Results
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object, _
10          MyQuery As Object, _
11          MyMetaCube As Object
12
13      'Excel Variables
14      Dim ReportRange As Range
15
16      'Other Variables
17      Dim MyData As Variant
18
19      Const MyFirstAttribute = "Brand"
20      Const MyMeasure = "Units Sold"
21
22  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
23      'Instantiate a Metabase object
24      Set MyMetabase = CreateObject("Metabase")
25
26      'Identify an ODBC Data Source
27      Let MyMetabase.ConnectString = "Metademo"
28
29      'Specify a set of metadata
30      Let MyMetabase.Name = "MetaCube Demo"
31
32      'Specify login to database
33      Let MyMetabase.Login = "metademo"
34      'Passes value to database on connection
35
36      'Specify database password
37      Let MyMetabase.Password = "Metademo"
38
39      'Log in to demonstration database, open DSS System
40      MyMetabase.Connect
41
42  'Define the Query
```

```
43   Set MyQuery = MyMetabase.Queries.Add("A New Query")
44   'Adds query to MyMetabase's collection of queries
45
46   MyQuery.QueryCategories.Add MyFirstAttribute
47
48   'Add measure to MyQuery's collection of measures
49   MyQuery.QueryItems.Add MyMeasure
50
51   'Display Query Definition
52   MsgBox MyQuery.SQL
53   'SQL property shows SQL generated for query before execution
54
55   'Get Query Results, Define Report
56   'Add cube to MyQuery's cube collection
57   Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
58
59   'Format data as an array VB can display, store in variable
60   Let MyData = MyMetaCube.ToVBArray
61   'The ToVBArray method implicitly requires MetaCube
62   'to execute the query on the relational database
63
64   'Clear "Query Report" Worksheet
65   Sheets("Query Report").Activate
66   Cells.Select
67   Selection.ClearContents
68
69   'Excel Code:  Defines Range of Cells, Presents Data in Worksheet
70   Worksheets.Item("Query Report").Activate
71   Set ReportRange = _
72       ActiveSheet.Range _
73       (ActiveSheet.Cells(1, 1), _
74       ActiveSheet.Cells _
75       (MyMetaCube.Rows, MyMetaCube.Columns))
76   Let ReportRange.Value = MyData
77   ReportRange.EntireColumn.AutoFit  'Sizes columns
78
79   End Sub
```

Explanation of Exercise 3

This exercise executes the query defined in the previous exercise and displays the data in an Excel spreadsheet named **Query Report**. If you have not identified a sheet in your active workbook by this name, follow the procedure outlined in "Renaming Worksheets and Macro Modules" on page 1-17.

Begin this procedure by declaring three new variables in lines 11 through 20: **MyMetaCube**, **MyData**, and **ReportRange**. The **MyMetaCube** variable stores an instance of the **MetaCube** class of objects that represents a particular configuration for the data retrieved from the relational database. The **MyData** variable stores the data retrieved from the query in a two-dimensional variant array that Excel can display in a spreadsheet. The **ReportRange** variable is a special Excel-type variable that represents the range of cells in a spreadsheet that the MetaCube analysis engine requires to display the data.

To bypass the task of declaring object variables, you can insert a file from the MetaCube library of training materials. Open the **M3_API.xls** workbook, select the tab labeled **Exercise #3**, and copy the appropriate variable declarations into your program.

Lines 22 to 40 establish a multidimensional connection to the relational database, and lines 42 to 55 define and display the query definition, as explained above. Line 57 instantiates a MetaCube object, which was compared in previous explanations to a report. More precisely, the MetaCube object represents the result set of a query, stored on the client as a virtual cube, on which you can readily perform operations. A Query object can own a collection of MetaCube objects, each defined to represent a query's data in a different format.

You instantiate a MetaCube object using syntax similar to the commands for instantiating QueryCategory and QueryItem objects, as discussed in the previous example. After you identify the collection of MetaCube objects belonging to **MyQuery**, you can add a new instance of a MetaCube object to this collection, passing as an argument a name for this virtual cube of data. This name does not refer in any way to MetaCube's metadata; like the query name argument, a MetaCube object's name is simply another way of distinguishing an item in a collection. Because you perform many operations on this virtual cube of data, we store this instantiation of the MetaCube object class in the object variable **MyMetaCube**. The command returns a value to an object variable and thus executes a function. Accordingly, the argument is enclosed in parentheses.

Instantiating the MetaCube object does not automatically execute the query on the relational database. To avoid premature execution of the query and to minimize client-server calls, the MetaCube analysis engine does not execute the query until the client application instructs the MetaCube analysis engine to perform some operation on the data. To explicitly command the MetaCube analysis engine to execute a query, use the **Retrieve** method of the **Queries** class of objects.

Line 60, which invokes the **ToVBArray** method of the MetaCube class of objects, converts the data represented by the virtual cube to an array, a function that can only be performed when the data has actually been returned to the client. For this reason, the **ToVBArray** method implicitly requires the MetaCube analysis engine to execute the query defined in lines 42 to 49. The variant variable **MyData** stores this data, automatically becoming an array variable of the appropriate dimensions. The **Let** syntax, though always unnecessary, is included throughout these exercises to emphasize the distinction between object variables and variables of other types.

Lines 65 to 77 activate a worksheet, define a range of cells into which you can import the array of data stored by the **MyData** variable, and assign the data in **MyData** to that range. The final line of this section adjusts the width of the report's columns so that none of the cells in the report are truncated. Since most of the code in this section corresponds to Excel rather than the MetaCube programming interface, you can copy this section of code from the **Exercise #3** worksheet in the **M3_API.xls** workbook, which is located in the exercises subdirectory of your MetaCube directory.

The only syntax of particular interest to the MetaCube developer in this section is the continuation of line 71, which extends to line 75. Here, the **Rows** and **Columns** properties of the **MetaCube** class of objects, which represent the number of rows and columns in the data within the cube, delineate the range of cells to reserve for the report. Insofar as both properties depend on the nature and amount of data retrieved, both can require MetaCube to execute a query. In this case, however, the **ToVBArray** method on line 60 already triggered the query's execution.

Exercise 4: Filtering the Query

In this exercise you deploy the Filter object to limit the range of data retrieved by the query.

```
1  Option Explicit 'All variables must be declared explicitly
2
3   'MetaCube API Exercise 4: Filtering the Query
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object, _
10          FilterFolder As Object, _
11          MyQuery As Object, _
12          MyMetaCube As Object
13
14      'Excel Variables
15      Dim ReportRange As Range
16
17      'Other Variables
18      Dim MyData As Variant
19
20      Const MyFirstAttribute = "Brand"
21      Const MyMeasure = "Units Sold"
22      Const MySavedFilter = "Boston"
23
24  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
25      'Instantiate a Metabase object
26      Set MyMetabase = CreateObject("Metabase")
27
28      'Identify an ODBC Data Source
29      Let MyMetabase.ConnectString = "Metademo"
30
31      'Specify a set of metadata
32      Let MyMetabase.Name = "MetaCube Demo"
33
34      'Specify login to database
35      Let MyMetabase.Login = "metademo"
36      'Passes value to database on connection
37
38      'Specify database password
39      Let MyMetabase.Password = "Metademo"
40
41      'Log in to demonstration database, open DSS System
42      MyMetabase.Connect
43
```

```
'Identify folder containing filters
 Set FilterFolder = MyMetabase.RootFolder. _
     Folders.Item("Public Filters")

 'Define the Query
 Set MyQuery = MyMetabase.Queries.Add("A New Query")
 'Adds query to MyMetabase's collection of queries

 MyQuery.QueryCategories.Add MyFirstAttribute

 'Add measure to MyQuery's collection of measures
 MyQuery.QueryItems.Add MyMeasure

 'Add saved filter; specifies filter name, who saved, and
folder
 MyQuery.Filters.AddSaved _
     MySavedFilter, "METAPUB", FilterFolder

 'Display Query Definition
 MsgBox MyQuery.SQL
 'SQL property shows SQL generated for query before execution

 'Get Query Results, Define Report
 'Add cube to MyQuery's cube collection
 Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")

 'Format data as an array VB can display, store in variable
 Let MyData = MyMetaCube.ToVBArray
 'The ToVBArray method implicitly requires MetaCube
 'to execute the query on the relational database

 'Clear "Query Report" Worksheet
 Sheets("Query Report").Activate
 Cells.Select
 Selection.ClearContents

 'Excel Code:  Defines Range of Cells, Presents Data in
Worksheet
 Worksheets.Item("Query Report").Activate
 Set ReportRange = _
     ActiveSheet.Range _
     (ActiveSheet.Cells(1, 1), _
     ActiveSheet.Cells _
     (MyMetaCube.Rows, MyMetaCube.Columns))
 Let ReportRange.Value = MyData
 ReportRange.EntireColumn.AutoFit  'Sizes columns

 End Sub
```

Explanation of Exercise 4

This exercise applies a saved filter to the previously defined query, limiting the range of data retrieved to two 4-week periods. Previously, the query had been completely unfiltered, returning brand sales for all dates recorded in the demonstration database. Now the query only returns the most recent 13 weeks of brand sales. The format of the resulting report remains the same, but the numeric values within the report decrease, since the sales for all time are necessarily less than the sales for 13 weeks.

Although MetaCube Explorer and MetaCube for Excel require you to filter on time, such requirements are artificially imposed by these applications to prevent users from issuing unconstrained queries. The MetaCube analysis engine allows you to submit for execution any query defined by valid measures, attributes, or even dimension elements.

The definitions of saved filters are stored in MetaCube's metadata. To access a saved filter, you must correctly identify the name of a filter associated with the DSS System you have opened, as well as the user name and folder under which that filter was saved. Security measures that prevent users of MetaCube Explorer and MetaCube for Excel from accessing filters saved by other users do not apply, since this requirement too is imposed by the application.

Line 22 declares a constant, **MySavedFilter**, which identifies the name of the filter saved in the metadata as **Boston**. This predefined, public filter installs with the demonstration database, limiting the data retrieved to the 13 most current weeks recorded therein.

To access this filter in the demonstration database, you identify the folder with which the Filter object is associated. MetaCube features a hierarchical folder interface for Query and Filter objects stored in the metadata tables of the relational database. Folders are descended from the **RootFolder** object, which is owned directly by the Metabase object. Each folder can, in turn, own subfolders. Lines 45 and 46 store the **Public Filters** folder in the **FilterFolder** object variable. For a full description of folder functionality, see "The Folders Class of Objects" on page 7-3.

The procedure executes as before until line 58, which instantiates a Filter object as a member of **MyQuery**'s collection of filters. As illustrated in Figure 1-2 on page 1-9, each Filter object belongs to a collection owned by a particular Query object.

Each collection of Filter objects can consist of both new filters and saved filters. For this reason, you must specify one of two methods to instantiate a Filter object: **AddSaved** or **AddNew**. Instantiating a new Filter object with the **AddNew** method requires you to subsequently define the filter's components.

The **AddSaved** method requires three arguments, the name of the filter, the user who created the filter, and the folder under which it was saved. Each Metabase object owns a **RootFolder**, which in turn can own a collection of subdirectories or folders, providing a hierarchical interface for logical objects stored in MetaCube's metadata. For more information about folders, see "The Folders Class of Objects" on page 7-3. In this case, **MySavedFilter**, represents the name of the saved filter, **metapub** identifies the user who defined this filter originally, and the **RootFolder** object, included here as an argument, indicates that the filter was originally saved in the root folder.

Exercise 5: Building a More Sophisticated Query, Pivoting

This exercise adds two attributes to the query's definition, displaying each value of one of the attributes in a separate column rather than in a separate row, as before.

```
1  Option Explicit 'All variables must be declared explicitly
2
3   'MetaCube API Exercise 5:Building a More Sophisticated Query,
   Pivoting
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object, _
10          FilterFolder As Object, _
11          MyQuery As Object, _
12          MyMetaCube As Object, _
13         MySummaryCategory As Object, _
14         MySortCategory As Object, _
15         MyPivotCategory As Object
16
17      'Excel Variables
18      Dim ReportRange As Range
19
20      'Other Variables
21      Dim MyData As Variant
22
23     Const OrientationColumn = 2
24      Const MyFirstAttribute = "Brand"
25      Const MySecondAttribute = "Region"
26     Const MyThirdAttribute = "Fiscal Week"
27      Const MyMeasure = "Units Sold"
28      Const MySavedFilter = "Boston"
29
30  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
31      'Instantiate a Metabase object
32      Set MyMetabase = CreateObject("Metabase")
33
34      'Identify an ODBC Data Source
35      Let MyMetabase.ConnectString = "Metademo"
36
37      'Specify a set of metadata
38      Let MyMetabase.Name = "MetaCube Demo"
39
40      'Specify login to database
41      Let MyMetabase.Login = "metademo"
```

```
42        'Passes value to database on connection
43
44        'Specify database password
45        Let MyMetabase.Password = "Metademo"
46
47        'Log in to demonstration database, open DSS System
48        MyMetabase.Connect
49
50    'Identify folder containing filters
51    Set FilterFolder = MyMetabase.RootFolder. _
52        Folders.Item("Public Filters")
53
54    'Define the Query
55    Set MyQuery = MyMetabase.Queries.Add("A New Query")
56    'Adds query to MyMetabase's collection of queries
57
58    Set MySummaryCategory = _
59        MyQuery.QueryCategories.Add(MyFirstAttribute)
60
61    Set MySortCategory = MyQuery.QueryCategories.Add _
62        (MySecondAttribute)
63
64    Set MyPivotCategory = _
65        MyQuery.QueryCategories.Add(MyThirdAttribute)
66    'Pivot this attribute to the column orientation
67    Let MyPivotCategory.Orientation = OrientationColumn
68
69    'Add measure to MyQuery's collection of measures
70    MyQuery.QueryItems.Add MyMeasure
71
72    'Add saved filter; specifies filter name, who saved, and
    folder
73    MyQuery.Filters.AddSaved _
74        MySavedFilter, "METAPUB", FilterFolder
75
76    'Display Query Definition
77    MsgBox MyQuery.SQL
78    'SQL property shows SQL generated for query before execution
79
80    'Get Query Results, Define Report
81    'Add cube to MyQuery's cube collection
82    Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
83
84    'Format data as an array VB can display, store in variable
85    Let MyData = MyMetaCube.ToVBArray
86    'The ToVBArray method implicitly requires MetaCube
87    'to execute the query on the relational database
88
89    'Clear "Query Report" Worksheet
```

```
90   Sheets("Query Report").Activate
91   Cells.Select
92   Selection.ClearContents
93
94   'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
95   Worksheets.Item("Query Report").Activate
96   Set ReportRange = _
97       ActiveSheet.Range _
98       (ActiveSheet.Cells(1, 1), _
99       ActiveSheet.Cells _
100      (MyMetaCube.Rows, MyMetaCube.Columns))
101  Let ReportRange.Value = MyData
102  ReportRange.EntireColumn.AutoFit  'Sizes columns
103
104  End Sub
```

Explanation of Exercise 5

In this procedure you add two attributes to the query definition, one of which is pivoted to columns. Each value of an attribute organized by columns defines a separate column in the resulting report. By default, all attributes are organized by rows.

As always, you begin by declaring any new variables and constants. Because you will perform subsequent operations on both the existing QueryCategory object as well the newly instantiated QueryCategory objects, it is convenient to store each instantiation in a new object variable, declared as **MySummary-Category**, **MySortCategory**, and **MyPivotCategory** in lines 13 through 15. In lines 25 and 26, you declare two additional constants, both of which refer to attributes defined in MetaCube's metadata. As before, declaring constants allows you to change one of the parameters of a query simply by changing the constant declaration, as opposed to replacing every reference to that attribute in the code that follows.

Line 23 also declares a constant, but for a different reason. Many MetaCube commands require numeric arguments whose significance, while fully documented in this reference, are not immediately clear. Substituting an aptly named constant for the rather cryptic numeric argument can make your code more readable and easier to debug. In this exercise, a numeric argument, 2, specifies an orientation by columns, which you substitute with the constant **OrientationColumn** whenever the argument is called for.

A complete list of this and similar constant declarations is provided in the file, **MetaCons.bas**, which installs in your MetaCube directory. Constant declarations for C++ developers can be found in **MetaCons.h**, in the same directory. The names of these constants are included in all documentation references to MetaCube's numeric arguments. The set of constant and variable declarations for this particular exercise can be copied from the tab labeled **Exercise #5** in the **M3_API.xls** workbook.

Line 58 modifies the code that instantiates the first QueryCategory, now storing the object in the object variable **MySummaryCategory**. As a consequence of returning a value to an object variable, the argument for this QueryCategory must be enclosed in parentheses. Lines 61 through 65 add a second and a third attribute to the query definition, instantiating QueryCategory objects in exactly the same manner as **MySummaryCategory**.

As defined, the query now consists of three attributes: **Brand**, **Region**, and **Fiscal Week**, all of which would normally be organized in rows and subrows, depending on their order of instantiation.

To organize an attribute by columns, you must assign a numeric value to the **Orientation** property of the QueryCategory object, as shown on line 67. Assigning a new value to the **Orientation** property of the QueryCategory object after the query has processed would require you to instantiate a new MetaCube object, but does involve re-querying the database. The value assigned to the **Orientation** property specifies the configuration of the QueryCategory, where 2 corresponds to pivoting by column, and 3 by page. In this example, the constant **OrientationColumn** substitutes for 2, an argument that is practically indecipherable without a reference. The resulting report displays brands and regions in rows, and weeks in columns. An upcoming exercise compares each column of data with its predecessor, calculating the difference between the two.

Exercise 6: Sorting by an Attribute

In this exercise you organize the values of an attribute in reverse-alphabetical order.

```
Option Explicit 'All variables must be declared explicitly

'MetaCube API Exercise 6: Sorting by an Attribute

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
         MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range

     'Other Variables
     Dim MyData As Variant

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2

     Const MyFirstAttribute = "Brand"
     Const MySecondAttribute = "Region"
     Const MyThirdAttribute = "Fiscal Week"
     Const MyMeasure = "Units Sold"
     Const MySavedFilter = "Boston"

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

      'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"

     'Specify a set of metadata
     Let MyMetabase.Name = "MetaCube Demo"

     'Specify login to database
     Let MyMetabase.Login = "metademo"
```

```
44      'Passes value to database on connection
45
46      'Specify database password
47      Let MyMetabase.Password = "Metademo"
48
49      'Log in to demonstration database, open DSS System
50      MyMetabase.Connect
51
52   'Identify folder containing filters
53   Set FilterFolder = MyMetabase.RootFolder. _
54       Folders.Item("Public Filters")
55
56   'Define the Query
57   Set MyQuery = MyMetabase.Queries.Add("A New Query")
58   'Adds query to MyMetabase's collection of queries
59
60   Set MySummaryCategory = _
61       MyQuery.QueryCategories.Add(MyFirstAttribute)
62
63   Set MySortCategory = MyQuery.QueryCategories.Add _
64       (MySecondAttribute)
65   'Sort the values of this attribute in reverse-alphabetical
   order
66   Let MySortCategory.SortDirection = _
67       SortDirectionDesc
68
69   Set MyPivotCategory = _
70       MyQuery.QueryCategories.Add(MyThirdAttribute)
71   'Pivot this attribute to the column orientation
72   Let MyPivotCategory.Orientation = OrientationColumn
73
74   'Add measure to MyQuery's collection of measures
75   MyQuery.QueryItems.Add MyMeasure
76
77   'Add saved filter; specifies filter name, who saved, and
   folder
78   'MyQuery.Filters.AddSaved _
79   '     MySavedFilter, "METAPUB", FilterFolder
80
81   'Display Query Definition
82   MsgBox MyQuery.SQL
83   'SQL property shows SQL generated for query before execution
84
85   'Get Query Results, Define Report
86   'Add cube to MyQuery's cube collection
87   Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
88
89   'Format data as an array VB can display, store in variable
90   Let MyData = MyMetaCube.ToVBArray
```

```
91   'The ToVBArray method implicitly requires MetaCube
92   'to execute the query on the relational database
93
94   'Clear "Query Report" Worksheet
95   Sheets("Query Report").Activate
96   Cells.Select
97   Selection.ClearContents
98
99   'Excel Code:  Defines Range of Cells, Presents Data in
     Worksheet
100  Worksheets.Item("Query Report").Activate
101  Set ReportRange = _
102      ActiveSheet.Range _
103      (ActiveSheet.Cells(1, 1), _
104      ActiveSheet.Cells _
105      (MyMetaCube.Rows, MyMetaCube.Columns))
106  Let ReportRange.Value = MyData
107  ReportRange.EntireColumn.AutoFit  'Sizes columns
108
109  End Sub
```

Explanation of Exercise 6

Sorting allows you to set the order in which the values of an attribute or a measure appear in a report. MetaCube can sort string values alphabetically and numbers from large to small, or vice versa. By default a report is sorted by the values of the attributes organized in rows and columns, in ascending order. Attributes organized by subrows and subcolumns are sorted after attributes organized by rows and columns are sorted, and only within each grouping.

Because each record consists of both attributes and measures, you cannot simultaneously sort on attributes organized by row and on measures. MetaCube automatically sorts attributes organized by row in ascending order and does not sort measures by default. Any sort that you apply on measures is likely to change the format of the report.

You can reverse the order of a sort on a QueryCategory by changing the value of that object's **SortDirection** property. To sort on a column of numeric data in a report, you must assign a value to the **SortColumn** property of the MetaCube object, as documented in "Sorting: SortDirection and SortColumn Properties" on page 8-53.

As a substitute for the cryptic values stored by both properties, declare an intuitively named constant in line 24, **SortDirectionDesc**, assigning it a value of 2 for a descending sort. As before, this constant declaration could have been taken from the **MetaCons.bas** file in your MetaCube directory.

Lines 34 through 50 establish a multidimensional connection to the relational database, as before. As you define the query in the ensuing section, you append lines 66 and 67, in which you assign the value represented by the **SortDirectionDesc** constant to the **SortDirection** property of **MySortCategory**.

Since **SortDirectionDesc** equals 2, MetaCube arranges the values of the **Region** attribute in a descending order, that is, reverse-alphabetically.

Exercise 7: Calculating Absolute Change

In this exercise, you add a calculated measure to the query definition, displaying the difference between every two columns of raw data in an interpolated column. The function for calculating the difference between the two columns is drawn from the main MetaCube snap-in or extension, **MCPlgMn**, which the MetaCube installation program enables, making it available in all application development environments. See "The Extensions Class of Objects" on page 5-3 for more details on extensions.

```
Option Explicit 'All variables must be declared explicitly

'MetaCube API Exercise 7: Calculating Absolute Change

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
         MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range

     'Other Variables
     Dim MyData As Variant

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2

     Const MyFirstAttribute = "Brand"
     Const MySecondAttribute = "Region"
     Const MyThirdAttribute = "Fiscal Week"
     Const MyMeasure = "Units Sold"
     Const MySavedFilter = "Boston"

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"
```

```
38
39        'Specify a set of metadata
40        Let MyMetabase.Name = "MetaCube Demo"
41
42        'Specify login to database
43        Let MyMetabase.Login = "metademo"
44        'Passes value to database on connection
45
46        'Specify database password
47        Let MyMetabase.Password = "Metademo"
48
49        'Log in to demonstration database, open DSS System
50        MyMetabase.Connect
51
52    'Identify folder containing filters
53    Set FilterFolder = MyMetabase.RootFolder. _
54        Folders.Item("Public Filters")
55
56    'Define the Query
57    Set MyQuery = MyMetabase.Queries.Add("A New Query")
58    'Adds query to MyMetabase's collection of queries
59
60    Set MySummaryCategory = _
61        MyQuery.QueryCategories.Add(MyFirstAttribute)
62
63    Set MySortCategory = MyQuery.QueryCategories.Add _
64        (MySecondAttribute)
65    'Sort the values of this attribute in reverse-alphabetical
  order
66    Let MySortCategory.SortDirection = _
67         SortDirectionDesc
68
69    Set MyPivotCategory = _
70        MyQuery.QueryCategories.Add(MyThirdAttribute)
71    'Pivot this attribute to the column orientation
72    Let MyPivotCategory.Orientation = OrientationColumn
73
74    'Add measure to MyQuery's collection of measures
75    MyQuery.QueryItems.Add MyMeasure
76
77    'Add second measure, on which to perform calculation
78    MyQuery.QueryItems.Add _
79       "Abs_Change(" + MyMeasure + ")"
80
81    'Add saved filter; specifies filter name, who saved, and
  folder
82    MyQuery.Filters.AddSaved _
83        MySavedFilter, "METAPUB", FilterFolder
84
```

```
'Display Query Definition
 MsgBox MyQuery.SQL
 'SQL property shows SQL generated for query before execution

 'Get Query Results, Define Report
 'Add cube to MyQuery's cube collection
 Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")

 'Format data as an array VB can display, store in variable
 Let MyData = MyMetaCube.ToVBArray
 'The ToVBArray method implicitly requires MetaCube
 'to execute the query on the relational database

 'Clear "Query Report" Worksheet
 Sheets("Query Report").Activate
 Cells.Select
 Selection.ClearContents

 'Excel Code:  Defines Range of Cells, Presents Data in
Worksheet
 Worksheets.Item("Query Report").Activate
 Set ReportRange = _
     ActiveSheet.Range _
     (ActiveSheet.Cells(1, 1), _
     ActiveSheet.Cells _
     (MyMetaCube.Rows, MyMetaCube.Columns))
 Let ReportRange.Value = MyData
 ReportRange.EntireColumn.AutoFit  'Sizes columns

 End Sub
```

Explanation of Exercise 7

MetaCube supports sophisticated comparison calculations, such as percent of total, moving averages, and quantiles. In this procedure, MetaCube evaluates the difference in sales from one week to the next. For each column of raw data in your report, the difference between that column and the preceding column of raw data is calculated and displayed in an interpolated third column.

The syntax for each calculation varies from function to function, depending on the number of arguments required by that function. Because the function itself returns values that MetaCube incorporates into the result as a QueryItem, you must enclose the entire expression in quotation marks, and the arguments required by the function in parentheses, as shown on lines 78 and 79. The plus symbols are required to concatenate the constant name with the rest of the string expression.

The only argument required by the absolute change function is the measure on which the calculation is performed. This function evaluates data as it changes from column to column. Other functions operate on data as it changes from row to row.

You could have based the calculation on a measure otherwise excluded from the report, displaying the number of units sold each week, as well the change in gross revenues or incurred costs from week to week. Such calculations prompt MetaCube to generate SQL that retrieves from the database the numeric necessary to perform the calculation, while only displaying the result of that calculation.

While most calculations are performed by functions included in MetaCube's main extension, the Summary object performs subtotals and other similar calculations. Subtotals are the subject of Exercise 8.

Exercise 8: Subtotals

In this exercise you use the Summary object to calculate subtotals in a report.

```
Option Explicit 'All variables must be declared explicitly

 'MetaCube API Exercise 8: Subtotals

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
   MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range

     'Other Variables
     Dim MyData As Variant

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2
     Const SummaryTotal = 1

     Const MyFirstAttribute = "Brand"
     Const MySecondAttribute = "Region"
     Const MyThirdAttribute = "Fiscal Week"
     Const MyMeasure = "Units Sold"
     Const MySavedFilter = "Boston"

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"

     'Specify a set of metadata
     Let MyMetabase.Name = "MetaCube Demo"

     'Specify login to database
     Let MyMetabase.Login = "metademo"
     'Passes value to database on connection
```

```
46
47        'Specify database password
48        Let MyMetabase.Password = "Metademo"
49
50        'Log in to demonstration database, open DSS System
51        MyMetabase.Connect
52
53    'Identify folder containing filters
54    Set FilterFolder = MyMetabase.RootFolder. _
55        Folders.Item("Public Filters")
56
57    'Define the Query
58    Set MyQuery = MyMetabase.Queries.Add("A New Query")
59    'Adds query to MyMetabase's collection of queries
60
61    Set MySummaryCategory = _
62        MyQuery.QueryCategories.Add(MyFirstAttribute)
63
64    Set MySortCategory = MyQuery.QueryCategories.Add _
65        (MySecondAttribute)
66    'Sort the values of this attribute in reverse-alphabetical
  order
67    Let MySortCategory.SortDirection = _
68    SortDirectionDesc
69
70    Set MyPivotCategory = _
71        MyQuery.QueryCategories.Add(MyThirdAttribute)
72    'Pivot this attribute to the column orientation
73    Let MyPivotCategory.Orientation = OrientationColumn
74
75    'Add measure to MyQuery's collection of measures
76    MyQuery.QueryItems.Add MyMeasure
77
78    'Add second measure, on which to perform calculation
79    MyQuery.QueryItems.Add _
80    "Abs_Change(" + MyMeasure + ")"
81
82    'Add saved filter; specifies filter name, who saved, and
  folder
83    MyQuery.Filters.AddSaved _
84        MySavedFilter, "METAPUB", FilterFolder
85
86    'Display Query Definition
87    MsgBox MyQuery.SQL
88    'SQL property shows SQL generated for query before execution
89
90    'Get Query Results, Define Report
91    'Add cube to MyQuery's cube collection
92    Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
```

```
93
94   'Subtotal: Regional sales for each brand
95   MyMetaCube.Summaries.Add _
96       MySummaryCategory, SummaryTotal
97
98   'Format data as an array VB can display, store in variable
99   Let MyData = MyMetaCube.ToVBArray
100  'The ToVBArray method implicitly requires MetaCube
101  'to execute the query on the relational database
102
103  'Clear "Query Report" Worksheet
104  Sheets("Query Report").Activate
105  Cells.Select
106  Selection.ClearContents
107
108  'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
109  Worksheets.Item("Query Report").Activate
110  Set ReportRange = _
111      ActiveSheet.Range _
112      (ActiveSheet.Cells(1, 1), _
113      ActiveSheet.Cells _
114      (MyMetaCube.Rows, MyMetaCube.Columns))
115  Let ReportRange.Value = MyData
116  ReportRange.EntireColumn.AutoFit  'Sizes columns
117
118  End Sub
```

Explanation of Exercise 8

MetaCube groups information by different attribute values, so you can calculate subtotals for each grouping. In the report generated by this procedure, MetaCube returns a record for each brand's sales in each region, grouping regional brand sales by brand. To calculate a brand's total sales in all regions, you sum every region's brands sales for that particular brand. In this case, we perform a subtotal by brand. For each brand, MetaCube interpolates a row representing that brand's total sales for all regions.

The subtotal calculation is performed on an existing set of data by instantiating an object within a collection belonging to the MetaCube object. For subtotals, as well as for averages, minimums, maximums, and counts and grand totals, you instantiate an object of the **Summaries** class, which descends from the **MetaCube** class of objects.

Both the QueryCategory on which the calculation is performed and the type of calculation to be performed depend on the arguments included in the command that instantiates the Summary object. On line 25, you declare a constant, **SummaryTotal**, to store the numeric argument that directs the Summary object to calculate subtotals. Other arguments direct the object to calculate averages, counts, minimums, and maximums, both for each brand and for the entire report. See Figure 8-26 on page 8-77.

Lines 95 and 96 instantiate the Summary object, specifying **MyMetaCube's** collection of Summary objects and deploying the general **Add** method. Two arguments follow, the first identifying the QueryCategory object on which to perform the calculation, the second indicating the type of calculation to perform. The first argument requires you to specify the actual QueryCategory object, rather than simply identifying the object by name. For this reason, you store all QueryCategory objects in object variables.

You have now completed many of the standard query operations. Exercises appearing later in this manual demonstrate the programming interface for more complex query operations such as buckets, comparisons, multi-fact table queries, background query processing, and parameterized filters.

The remainder of the tutorial discusses the construction of a simple query interface for defining ad hoc queries.

Exercise 9: Building an Interface

In this exercise you populate several list boxes with the names of all the attributes available for querying in the DSS System to which you have connected.

```
Option Explicit 'All variables must be declared explicitly

'MetaCube API Exercise 9: Building an Interface

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
         MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range, _
        MyListBox As ListBox

     'Other Variables
     Dim MyData As Variant, _
         ArrayofItems As Variant, _
        DimensionCount As Integer

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2
     Const SummaryTotal = 1
    Const DisplayStyleQuery = 2

     Const MyFirstAttribute = "Brand"
     Const MySecondAttribute = "Region"
     Const MyThirdAttribute = "Fiscal Week"
     Const MyMeasure = "Units Sold"
     Const MySavedFilter = "Boston"

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"
```

```
43
44        'Specify a set of metadata
45        Let MyMetabase.Name = "MetaCube Demo"
46
47        'Specify login to database
48        Let MyMetabase.Login = "metademo"
49        'Passes value to database on connection
50
51        'Specify database password
52        Let MyMetabase.Password = "Metademo"
53
54        'Log in to demonstration database, open DSS System
55        MyMetabase.Connect
56
57    'Identify folder containing filters
58    Set FilterFolder = MyMetabase.RootFolder. _
59        Folders.Item("Public Filters")
60
61    'Query Interface:  Attributes
62       Worksheets.Item("Define the Query").Activate
63
64    'Cycle through DSS System's dimensions
65    For DimensionCount = 0 To _
66                MyMetabase.Dimensions.Count - 1
67
68        'Get array of attributes
69        Let ArrayofItems = _
70          MyMetabase.Dimensions.Item(DimensionCount). _
71          AttributeNames(DisplayStyleQuery).ArrayValues
72
73        'Create listboxes
74        Set MyListBox = ActiveSheet.ListBoxes.Add _
75        ((DimensionCount * 80), 50, 70, 100)
76        'A list box for each dimension, each further to the right
77
78        'Populates each list box w/ attributes of a dimension
79        MyListBox.AddItem ArrayofItems
80
81    Next DimensionCount
82
83    'Define the Query
84    Set MyQuery = MyMetabase.Queries.Add("A New Query")
85    'Adds query to MyMetabase's collection of queries
86
87    Set MySummaryCategory = _
88        MyQuery.QueryCategories.Add(MyFirstAttribute)
89
90    Set MySortCategory = MyQuery.QueryCategories.Add _
91        (MySecondAttribute)
92    'Sort the values of this attribute in reverse-alphabetical
    order
93    Let MySortCategory.SortDirection = SortDirectionDesc
94
```

```
95   Set MyPivotCategory = _
96       MyQuery.QueryCategories.Add(MyThirdAttribute)
97   'Pivot this attribute to the column orientation
98   Let MyPivotCategory.Orientation = OrientationColumn
99
100  'Add measure to MyQuery's collection of measures
101  MyQuery.QueryItems.Add MyMeasure
102
103  'Add second measure, on which to perform calculation
104  MyQuery.QueryItems.Add _
105      "Abs_Change(" + MyMeasure + ")"
106
107  'Add saved filter; specifies filter name, who saved, and
     folder
108  MyQuery.Filters.AddSaved _
109      MySavedFilter, "METAPUB", FilterFolder
110
111  'Display Query Definition
112  MsgBox MyQuery.SQL
113  'SQL property shows SQL generated for query before execution
114
115  'Get Query Results, Define Report
116  'Add cube to MyQuery's cube collection
117  Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
118
119  'Subtotal: Regional sales for each brand
120  MyMetaCube.Summaries.Add _
121      MySummaryCategory, SummaryTotal
122
123  'Format data as an array VB can display, store in variable
124  Let MyData = MyMetaCube.ToVBArray
125  'The ToVBArray method implicitly requires MetaCube
126  'to execute the query on the relational database
127
128  'Clear "Query Report" Worksheet
129  Sheets("Query Report").Activate
130  Cells.Select
131  Selection.ClearContents
132
133  'Excel Code:  Defines Range of Cells, Presents Data in
     Worksheet
134  Worksheets.Item("Query Report").Activate
135  Set ReportRange = _
136      ActiveSheet.Range _
137      (ActiveSheet.Cells(1, 1), _
138      ActiveSheet.Cells _
139      (MyMetaCube.Rows, MyMetaCube.Columns))
140  Let ReportRange.Value = MyData
141  ReportRange.EntireColumn.AutoFit  'Sizes columns
```

```
142
143   End Sub
```

Explanation of Exercise 9

This exercise and the two exercises that follow create a simple query interface, displaying lists of available attributes, measures, and saved filters. While you build this interface, the application continues to execute the query you defined in the previous exercises. Ultimately, however, you can prompt the user to enter the names of attributes, measures, and filters to define the query.

This exercise populates a set of list boxes with the attributes for each dimension in the **MetaCube Demo** DSS System, the first step in building an interface for designing a query. Building even a simple interface requires extensive deployment of Excel-specific objects, properties, and methods. For this reason, you might want to use the code provided in the tab labeled **Exercise #9** in the **M3_API.xls** workbook, installed in the exercises subdirectory of your MetaCube directory.

Lines 23 and 24 declare two variables, **ArrayofItems** and **DimensionCount**. In this exercise, the **ArrayofItems** variant variable stores an array of attribute names that MetaCube retrieves from the metadata and displays in a list box for each dimension. The **DimensionCount** integer variable serves as a counter in a For... Next loop that cycles through each dimension in the DSS System. This loop does not, however, count through each attribute for each dimension, as the names of the attributes associated with a particular dimension can be retrieved together as an array.

A For... Next loop repeats an action or series of actions a set number of times. In this exercise, the number of repetitions is determined by the number of dimensions in the DSS System, as represented by the Metabase object **MyMetabase**. **MyMetabase** owns a collection of dimensions, which has, as a collection, a **Count** property. This collection was not pictured in the simplified diagram of Figure 1-2 on page 1-9. In line 66, the **Count** property returns the number of items within the collection: in this case, the number of dimensions in **MyMetabase**'s collection. The metadata for this DSS System, downloaded from the relational database upon connection, populates this collection with a set of Dimension objects. Since the **DimensionCount** loop counter begins at 0, you must subtract 1 from the number of dimensions, because their count begins at 1. This ensures that the loop repeats only once for each dimension.

For each dimension, the loop performs three tasks:

- Retrieves the names of that dimension's attributes identified as valid for use in queries
- Creates a separate list box, positioning each new list box progressively further to the right
- Populates the list box with the names of the attributes

Lines 69 through 71 retrieve, as an array, the attributes for a particular dimension. The latter half of this complicated equation can best be understood in parts.

The first part of the equation, **MyMetabase.Dimensions.Item** (**DimensionsCount**), specifies the dimension for which MetaCube will retrieve an array of attributes. The **MyMetabase** instantiation of the **Metabase** class of objects represents a particular multidimensional view of relational data, as defined by the **MetaCube Demo** DSS System.

Like any Metabase object, **MyMetabase** owns a collection of dimensions and fact tables, the fundamental logical objects in any DSS System. Because a DSS System can associate different dimensions with different fact tables, each fact table also has a collection of dimensions, a subset of the Metabase object's collection that specifically correspond to the parent fact table.

The fact table at issue in the demonstration system, however, joins to all dimensions, so the DSS System's collection of Dimension objects contains the same items as the FactTable object's collection. You can thus safely refer to **MyMetabase**'s collection of dimensions without including a Dimension object associated with the wrong fact table.

On line 70, you identify the collection of dimensions owned by the Metabase object in the usual way, specifying first the parent and then the collection itself. The Dimension object within the collection is identified by index number.

Rather than providing a constant value for the index number, you can identify different dimensions by incrementing the index number at every cycle through the For... Next loop. With each iteration of the loop, the **DimensionCount** integer increments by one, identifying a different item within **MyMetabase**'s collection of dimensions.

The second part of the equation, **AttributeNames(DisplayStyle-Query).ArrayValues**, retrieves all valid attributes within the specified dimension, returning those values as an array. The **DisplayStyleQuery** argument, which corresponds to a numeric constant declared on line 29, specifies attributes that have been validated for use in queries. A different argument value specifies only those attributes validated for use in filters. When you create the metadata for an attribute, you indicate whether the attribute is valid for use in queries or filters or both. For an explanation of how to validate attributes for use in queries or filters through MetaCube's programming interface, see Figure 4-7 on page 4-18.

MetaCube initially returns the attribute name in a generic format, which the **ArrayValues** method converts to an array. For development environments that do not support arrays, such as Visual Basic 3.0, MetaCube can return value sets in different formats, such as tab-delimited strings. The **Array-ofItems** variable automatically sizes itself to store the values of the array.

Lines 74 and 75 create a list box in the active worksheet, **Define the Query**. This instance of the list box is stored in the MyListBox variable, which you declared in line 19 as a special Excel-type variable. Including the **Dimension-Count** variable as a parameter for positioning the list box moves the location of each new list box farther to the right as the loop counter increments. Line 79 displays the list of attribute names in the most recently created list box. For more information about creating, positioning, and populating list boxes, consult Excel's on-line Visual Basic for Applications Help.

The remainder of this procedure defines and executes the query as before.

Exercise 10: Creating and Populating a Measures List Box

In this exercise you populate a list box with the names of the measures available in the DSS System to which you have connected.

```
1  Option Explicit 'All variables must be declared explicitly
2
3  'MetaCube API Exercise 10: Creating and Populating a Measures
   ListBox
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object, _
10          FilterFolder As Object, _
11          MyQuery As Object, _
12          MyMetaCube As Object, _
13     MySummaryCategory As Object, _
14          MySortCategory As Object, _
15          MyPivotCategory As Object
16
17      'Excel Variables
18      Dim ReportRange As Range, _
19          MyListBox As ListBox
20
21      'Other Variables
22      Dim MyData As Variant, _
23          ArrayofItems As Variant, _
24          DimensionCount As Integer
25
26      Const OrientationColumn = 2
27      Const SortDirectionDesc = 2
28      Const SummaryTotal = 1
29      Const DisplayStyleQuery = 2
30
31      Const MyFirstAttribute = "Brand"
32      Const MySecondAttribute = "Region"
33      Const MyThirdAttribute = "Fiscal Week"
34      Const MyMeasure = "Units Sold"
35      Const MySavedFilter = "Boston"
36
37  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
38      'Instantiate a Metabase object
39      Set MyMetabase = CreateObject("Metabase")
40
41      'Identify an ODBC Data Source
42      Let MyMetabase.ConnectString = "Metademo"
```

```
43
44        'Specify a set of metadata
45        Let MyMetabase.Name = "MetaCube Demo"
46
47        'Specify login to database
48        Let MyMetabase.Login = "metademo"
49        'Passes value to database on connection
50
51        'Specify database password
52        Let MyMetabase.Password = "Metademo"
53
54        'Log in to demonstration database, open DSS System
55        MyMetabase.Connect
56
57    'Identify folder containing filters
58    Set FilterFolder = MyMetabase.RootFolder. _
59        Folders.Item("Public Filters")
60
61    'Query Interface:  Attributes
62        Worksheets.Item("Define the Query").Activate
63
64    'Cycle through DSS System's dimensions
65    For DimensionCount = 0 To _
66                MyMetabase.Dimensions.Count - 1
67
68        'Get array of attributes
69        Let ArrayofItems = _
70          MyMetabase.Dimensions.Item(DimensionCount). _
71          AttributeNames(DisplayStyleQuery).ArrayValues
72
73        'Create listboxes
74        Set MyListBox = ActiveSheet.ListBoxes.Add _
75        ((DimensionCount * 80), 50, 70, 100)
76       'A list box for each dimension, each further to the right
77
78        'Populates each list box w/ attributes of a dimension
79        MyListBox.AddItem ArrayofItems
80
81    Next DimensionCount
82
83    'Query Interface:  Measures
84
85    'Create one list box for measures
86    Set MyListBox = ActiveSheet.ListBoxes.Add _
87        ((DimensionCount * 80), 50, 70, 100)
88    'Previous "Next DimensionCount" added 1, moves listbox right
89
90    'Get array of available measure names
91    Let ArrayofItems = _
```

```
92        MyMetabase.FactTables. _
93        Item("Sales Transactions"). _
94        MeasureNames(DisplayStyleQuery).ArrayValues
95
96    'Populate listbox with array of measure names
97    MyListBox.AddItem ArrayofItems
98
99    'Define the Query
100   Set MyQuery = MyMetabase.Queries.Add("A New Query")
101   'Adds query to MyMetabase's collection of queries
102
103   Set MySummaryCategory = _
104       MyQuery.QueryCategories.Add(MyFirstAttribute)
105
106   Set MySortCategory = MyQuery.QueryCategories.Add _
107       (MySecondAttribute)
108   'Sort the values of this attribute in reverse-alphabetical
    order
109   Let MySortCategory.SortDirection = SortDirectionDesc
110
111   Set MyPivotCategory = _
112       MyQuery.QueryCategories.Add(MyThirdAttribute)
113   'Pivot this attribute to the column orientation
114   Let MyPivotCategory.Orientation = OrientationColumn
115
116   'Add measure to MyQuery's collection of measures
117   MyQuery.QueryItems.Add MyMeasure
118
119   'Add second measure, on which to perform calculation
120   MyQuery.QueryItems.Add _
121         "Abs_Change(" + MyMeasure + ")"
122
123   'Add saved filter; specifies filter name, who saved, and
    folder
124   MyQuery.Filters.AddSaved _
125       MySavedFilter, "METAPUB", FilterFolder
126
127   'Display Query Definition
128   MsgBox MyQuery.SQL
129   'SQL property shows SQL generated for query before execution
130
131   'Get Query Results, Define Report
132   'Add cube to MyQuery's cube collection
133   Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
134
135   'Subtotal: Regional sales for each brand
136   MyMetaCube.Summaries.Add _
137       MySummaryCategory, SummaryTotal
138
```

```
139   'Format data as an array VB can display, store in variable
140   Let MyData = MyMetaCube.ToVBArray
141   'The ToVBArray method implicitly requires MetaCube
142   'to execute the query on the relational database
143
144   'Clear "Query Report" Worksheet
145   Sheets("Query Report").Activate
146   Cells.Select
147   Selection.ClearContents
148
149   'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
150   Worksheets.Item("Query Report").Activate
151   Set ReportRange = _
152       ActiveSheet.Range _
153       (ActiveSheet.Cells(1, 1), _
154       ActiveSheet.Cells _
155       (MyMetaCube.Rows, MyMetaCube.Columns))
156   Let ReportRange.Value = MyData
157   ReportRange.EntireColumn.AutoFit  'Sizes columns
158
159   End Sub
```

Explanation of Exercise 10

This exercise creates a list box to store the names of measures available in the data source, the **Sales Transactions** fact table. As before, the new code for this exercise relies heavily upon Excel functionality and can be copied from the tab labeled **Exercise #10** in the **M3_API.xls** workbook, located in your MetaCube training directory.

The syntax for populating a list box with measures is nearly identical to the code in lines 65 through 81, which displays the attributes for each dimension in separate listboxes. Consequently, you can reuse the variables and constants declared in the previous exercise.

In the previous exercise you cycled through every dimension in the DSS System to retrieve the names of valid query attributes, but to retrieve the names of measures you need merely access a single fact table's collection of measures. Although you can define a query retrieving information from multiple fact tables, such a task is more complicated and is left to advanced developers. This exercise creates a list box to display the valid query measures for one of the two fact tables in the **MetaCube Demo** DSS System, the **Sales Transaction** fact table.

Lines 83 and 87 create this list box, including the **DimensionCount** variable as an element in the set of arguments that determine its location. As the value of the **DimensionCount** variable increases, the position of the list box moves farther to the right. Because the last line of the loop defined in the previous exercise increments the **DimensionsCount** loop counter before exiting the loop, the position of the left edge of the new measures list box appears 80 points to the right of the last dimension list box. Each point corresponds to 1/72nd of an inch.

Lines 91 through 94 store an array of measure names in the **ArrayofItems** variant variable. This complicated command parallels the structure of the command to retrieve a dimension's array of attribute names (lines 69 through 71). The command begins by identifying the particular fact table within a DSS System for which you want to retrieve measure names. Reference this collection item not by a variable index number, as you did when identifying dimensions, but rather by name.

All of the exercises in this tutorial have, by default, queried on measures in the fact table that you now explicitly identify as **Sales Transactions**. This fact table is the default because it was instantiated first when the metadata for this DSS System was created.

The **MeasureNames** property represents the names of all valid measures stored in a particular fact table. This property requires the same argument as the **AttributeNames** property, in which the **DisplayStyleQuery** constant limits the items in the array to those validated for use in a query. As before, the **ArrayValues** method returns the set of measure names as an array, which the variant variable, **ArrayofItems**, can easily accept. Line 97 populates the newly created list box with this array of measure names.

Exercise 11: Displaying a List of Saved Filters

In this exercise, you complete the interface begun in Exercise 9, displaying the names of all filters saved by the **metapub** user in a particular folder object.

```
Option Explicit 'All variables must be declared explicitly

'MetaCube API Exercise 11: Displaying a List of Saved Filters

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
   MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range, _
         MyListBox As ListBox

     'Other Variables
     Dim MyData As Variant, _
         ArrayofItems As Variant, _
         DimensionCount As Integer

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2
     Const SummaryTotal = 1
     Const DisplayStyleQuery = 2

     Const MyFirstAttribute = "Brand"
     Const MySecondAttribute = "Region"
     Const MyThirdAttribute = "Fiscal Week"
     Const MyMeasure = "Units Sold"
     Const MySavedFilter = "Boston"

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"

```

```
44      'Specify a set of metadata
45      Let MyMetabase.Name = "MetaCube Demo"
46
47      'Specify login to database
48      Let MyMetabase.Login = "metademo"
49      'Passes value to database on connection
50
51      'Specify database password
52      Let MyMetabase.Password = "Metademo"
53
54      'Log in to demonstration database, open DSS System
55      MyMetabase.Connect
56
57  'Identify folder containing filters
58  Set FilterFolder = MyMetabase.RootFolder. _
59      Folders.Item("Public Filters")
60
61  'Query Interface:  Attributes
62      Worksheets.Item("Define the Query").Activate
63
64  'Cycle through DSS System's dimensions
65  For DimensionCount = 0 To _
66              MyMetabase.Dimensions.Count - 1
67
68      'Get array of attributes
69      Let ArrayofItems = _
70        MyMetabase.Dimensions.Item(DimensionCount). _
71        AttributeNames(DisplayStyleQuery).ArrayValues
72
73      'Create listboxes
74      Set MyListBox = ActiveSheet.ListBoxes.Add _
75      ((DimensionCount * 80), 50, 70, 100)
76     'A list box for each dimension, each further to the right
77
78      'Populates each list box w/ attributes of a dimension
79      MyListBox.AddItem ArrayofItems
80
81  Next DimensionCount
82
83  'Query Interface:  Measures
84
85  'Create one list box for measures
86  Set MyListBox = ActiveSheet.ListBoxes.Add _
87      ((DimensionCount * 80), 50, 70, 100)
88  'Previous "Next DimensionCount" added 1, moves listbox right
89
90  'Get array of available measure names
91  Let ArrayofItems = _
92      MyMetabase.FactTables. _
```

```
93        Item("Sales Transactions"). _
94        MeasureNames(DisplayStyleQuery).ArrayValues
95
96    'Populate listbox with array of measure names
97    MyListBox.AddItem ArrayofItems
98
99    'Query Interface:  Saved Filters
100
101   'Create listbox for saved filters
102   Set MyListBox = ActiveSheet.ListBoxes.Add _
103       (((DimensionCount + 1) * 80), 50, 120, 150)
104   'Increments "DimensionCount" by 1, moves listbox to right
105
106   'Get array of filters in root folder owned by METAPUB
107   Let ArrayofItems = _
108       FilterFolder.FilterNames _
109       ("METAPUB", "").ArrayValues
110
111   'Populate list box with array of filter names
112   MyListBox.AddItem ArrayofItems
113
114   'Define the Query
115   Set MyQuery = MyMetabase.Queries.Add("A New Query")
116   'Adds query to MyMetabase's collection of queries
117
118   Set MySummaryCategory = _
119       MyQuery.QueryCategories.Add(MyFirstAttribute)
120
121   Set MySortCategory = MyQuery.QueryCategories.Add _
122       (MySecondAttribute)
123   'Sort the values of this attribute in reverse-alphabetical
      order
124   Let MySortCategory.SortDirection = SortDirectionDesc
125
126   Set MyPivotCategory = _
127       MyQuery.QueryCategories.Add(MyThirdAttribute)
128   'Pivot this attribute to the column orientation
129   Let MyPivotCategory.Orientation = OrientationColumn
130
131   'Add measure to MyQuery's collection of measures
132   MyQuery.QueryItems.Add MyMeasure
133
134   'Add second measure, on which to perform calculation
135   MyQuery.QueryItems.Add _
136       "Abs_Change(" + MyMeasure + ")"
137
138   'Add saved filter; specifies filter name, who saved, and
      folder
139   MyQuery.Filters.AddSaved _
```

```
140         MySavedFilter, "METAPUB", FilterFolder
141
142     'Display Query Definition
143     MsgBox MyQuery.SQL
144     'SQL property shows SQL generated for query before execution
145
146     'Get Query Results, Define Report
147     'Add cube to MyQuery's cube collection
148     Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
149
150     'Subtotal: Regional sales for each brand
151     MyMetaCube.Summaries.Add _
152         MySummaryCategory, SummaryTotal
153
154     'Format data as an array VB can display, store in variable
155     Let MyData = MyMetaCube.ToVBArray
156     'The ToVBArray method implicitly requires MetaCube
157     'to execute the query on the relational database
158
159     'Clear "Query Report" Worksheet
160     Sheets("Query Report").Activate
161     Cells.Select
162     Selection.ClearContents
163
164     'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
165     Worksheets.Item("Query Report").Activate
166     Set ReportRange = _
167         ActiveSheet.Range _
168         (ActiveSheet.Cells(1, 1), _
169         ActiveSheet.Cells _
170         (MyMetaCube.Rows, MyMetaCube.Columns))
171     Let ReportRange.Value = MyData
172     ReportRange.EntireColumn.AutoFit  'Sizes columns
173
174     End Sub
```

Explanation of Exercise 11

In this exercise, you display the names of publicly saved filters, completing the simple query interface. As before, you can incorporate variables declared for previous sections of the application, such as **MyListBox**, **ArrayofItems**, and **DimensionCount**. The new list box object, instantiated and stored in **MyListBox** on lines 102 and 103, is positioned to the right of previous list boxes. The **DimensionCount** variable, used throughout the application as a parameter for the horizontal positioning of list boxes, is artificially incremented by 1 from its previous value and multiplied by a factor of 80. It remains to retrieve the names of the filters stored in the root folder for this DSS System.

As discussed in "Explanation of Exercise 4" on page 2-19, MetaCube offers a hierarchical folder interface for storing logical objects as metadata. The **RootFolder** object owned by the Metabase object represents the first folder, and any Folder objects owned by **RootFolder** represent subdirectories of that first folder.

On lines 107 through 109 of this exercise, you display in a list box the names of filters saved by MetaCube's public user, **metapub**, in the folder identified by the FilterFolder object. The **FilterNames** method of any Folder object, including the **RootFolder** object, requires two arguments: the name of the user who saved the filters in question, and the name of the group to which those filters belong. Usually filters are organized by group according to the dimension with which they are associated. Passing an empty string for the second argument returns all filters, regardless of their groupings. Since the **FilterNames** method returns the names of all filters in a generic format, the **ArrayValues** method converts the value list to an array that is ultimately stored in the **ArrayofItems** variant variable.

Line 112 adds the items in this array to the list box defined in lines 102 and 103. The remainder of the procedure executes as before.

Exercise 12: Prompting Users to Define Queries

In this exercise you replace the constants that previously supplied the arguments for attribute, measure, and filter specifications with variables, the values of which can be supplied by a user.

```
1  Option Explicit 'All variables must be declared explicitly
2
3  'MetaCube API Exercise 12: Prompting Users to Define Queries
4
5   Sub MetaCube_API()
6
7   'Declare Variables and Constants
8       'Object Variables
9       Dim MyMetabase As Object, _
10          FilterFolder As Object, _
11          MyQuery As Object, _
12          MyMetaCube As Object, _
13     MySummaryCategory As Object, _
14          MySortCategory As Object, _
15          MyPivotCategory As Object
16
17      'Excel Variables
18      Dim ReportRange As Range, _
19          MyListBox As ListBox
20
21      'Other Variables
22      Dim MyData As Variant, _
23          ArrayofItems As Variant, _
24          DimensionCount As Integer, _
25         MyFirstAttribute As String, _
26         MySecondAttribute As String, _
27         MyThirdAttribute As String, _
28         MyMeasure As String, _
29         MySavedFilter As String
30
31      Const OrientationColumn = 2
32      Const SortDirectionDesc = 2
33      Const SummaryTotal = 1
34      Const DisplayStyleQuery = 2
35      'Const MyFirstAttribute = "Brand"
36      'Const MySecondAttribute = "Region"
37      'Const MyThirdAttribute = "Fiscal Week"
38      'Const MyMeasure = "Units Sold"
39      'Const MySavedFilter = "Last 13 Weeks"
40
41  'Create & Define A Metabase: Log in to RDBMS, Open DSS System
42      'Instantiate a Metabase object
```

```
43        Set MyMetabase = CreateObject("Metabase")
44
45        'Identify an ODBC Data Source
46        Let MyMetabase.ConnectString = "Metademo"
47
48        'Specify a set of metadata
49        Let MyMetabase.Name = "MetaCube Demo"
50
51        'Specify login to database
52        Let MyMetabase.Login = "metademo"
53        'Passes value to database on connection
54
55        'Specify database password
56        Let MyMetabase.Password = "Metademo"
57
58        'Log in to demonstration database, open DSS System
59        MyMetabase.Connect
60
61    'Identify folder containing filters
62    Set FilterFolder = MyMetabase.RootFolder. _
63        Folders.Item("Public Filters")
64
65   'Query Interface:  Attributes
66        Worksheets.Item("Define the Query").Activate
67
68    'Cycle through DSS System's dimensions
69    For DimensionCount = 0 To _
70                MyMetabase.Dimensions.Count - 1
71
72        'Get array of attributes
73        Let ArrayofItems = _
74          MyMetabase.Dimensions.Item(DimensionCount). _
75          AttributeNames(DisplayStyleQuery).ArrayValues
76
77        'Create listboxes
78        Set MyListBox = ActiveSheet.ListBoxes.Add _
79        ((DimensionCount * 80), 50, 70, 100)
80       'A list box for each dimension, each further to the right
81
82        'Populates each list box w/ attributes of a dimension
83        MyListBox.AddItem ArrayofItems
84
85    Next DimensionCount
86
87    'Query Interface:  Measures
88
89    'Create one list box for measures
90    Set MyListBox = ActiveSheet.ListBoxes.Add _
91        ((DimensionCount * 80), 50, 70, 100)
```

```
'Previous "Next DimensionCount" added 1, moves listbox right

'Get array of available measure names
Let ArrayofItems = _
    MyMetabase.FactTables. _
    Item("Sales Transactions"). _
    MeasureNames(DisplayStyleQuery).ArrayValues

'Populate listbox with array of measure names
MyListBox.AddItem ArrayofItems

'Query Interface:  Saved Filters

'Create listbox for saved filters
Set MyListBox = ActiveSheet.ListBoxes.Add _
    (((DimensionCount + 1) * 80), 50, 120, 150)
'Increments "DimensionCount" by 1, moves listbox to right

'Get array of filters in root folder owned by METAPUB
Let ArrayofItems = _
    FilterFolder.FilterNames _
    ("METAPUB", "").ArrayValues

'Populate list box with array of filter names
MyListBox.AddItem ArrayofItems

'Retrieve Names of Attributes, Measure, Filter from User
Let MyFirstAttribute = _
        InputBox(Prompt:= _
        "Enter an attribute on which to query:", _
        Title:="First Attribute")

Let MySecondAttribute = _
        InputBox(Prompt:= _
        "Enter the name of another attribute:", _
        Title:="Second Attribute")

Let MyThirdAttribute = _
        InputBox(Prompt:= _
        "Enter the name of a third attribute:", _
        Title:="Attribute Organized by Columns")

Let MyMeasure = _
        InputBox(Prompt:= _
        "Enter a measure on which to query:", _
        Title:="Measure")

Let MySavedFilter = _
        InputBox(Prompt:= _
```

```
141          "Enter the name of a saved filter:", _
142          Title:="Filter")
143
144   'Define the Query
145   Set MyQuery = MyMetabase.Queries.Add("A New Query")
146   'Adds query to MyMetabase's collection of queries
147
148   Set MySummaryCategory = _
149       MyQuery.QueryCategories.Add(MyFirstAttribute)
150
151   Set MySortCategory = MyQuery.QueryCategories.Add _
152       (MySecondAttribute)
153   'Sort the values of this attribute in reverse-alphabetical
    order
154   Let MySortCategory.SortDirection = SortDirectionDesc
155
156   Set MyPivotCategory = _
157       MyQuery.QueryCategories.Add(MyThirdAttribute)
158   'Pivot this attribute to the column orientation
159   Let MyPivotCategory.Orientation = OrientationColumn
160
161   'Add measure to MyQuery's collection of measures
162   MyQuery.QueryItems.Add MyMeasure
163
164   'Add second measure, on which to perform calculation
165   MyQuery.QueryItems.Add _
166       "Abs_Change(" + MyMeasure + ")"
167
168   'Add saved filter; specifies filter name, who saved, and
    folder
169   MyQuery.Filters.AddSaved _
170       MySavedFilter, "METAPUB", FilterFolder
171
172   'Display Query Definition
173   MsgBox MyQuery.SQL
174   'SQL property shows SQL generated for query before execution
175
176   'Get Query Results, Define Report
177   'Add cube to MyQuery's cube collection
178   Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
179
180   'Subtotal: Regional sales for each brand
181   MyMetaCube.Summaries.Add _
182       MySummaryCategory, SummaryTotal
183
184   'Format data as an array VB can display, store in variable
185   Let MyData = MyMetaCube.ToVBArray
186   'The ToVBArray method implicitly requires MetaCube
187   'to execute the query on the relational database
```

```
188
189   'Clear "Query Report" Worksheet
190   Sheets("Query Report").Activate
191   Cells.Select
192   Selection.ClearContents
193
194   'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
195   Worksheets.Item("Query Report").Activate
196   Set ReportRange = _
197       ActiveSheet.Range _
198       (ActiveSheet.Cells(1, 1), _
199       ActiveSheet.Cells _
200       (MyMetaCube.Rows, MyMetaCube.Columns))
201   Let ReportRange.Value = MyData
202   ReportRange.EntireColumn.AutoFit  'Sizes columns
203
204   End Sub
```

Explanation of Exercise 12

This exercise demonstrates how easily you can prompt users to define an ad hoc query. Because you have declared constants for each parameter in the query definition, you can replace those constants with string variables, requesting the user to enter the values for each. The existing structure of Metabase, Query, QueryCategory, QueryItem, MetaCube, and Summary objects remains intact; only the values of their arguments change, as well as the way in which the application provides those values. Consequently, this exercise does not introduce any MetaCube functionality, and all of the new code can be copied from the tab labeled **Exercise #12**, in the **M3_API.xls** workbook located in your training directory.

Lines 25 through 29 declare variables of the same names as the lined-out constants appearing on lines 35 through 39. These variables replace the constants of the same name; you should thus delete these constant declarations. The arguments represented by each variable are all of the string type. As in previous exercises, the procedure subsequently connects to the database and opens a particular multidimensional view of the tables and columns, displaying in list boxes the library of attributes, measures, and saved filters that a user can incorporate into his or her query definition.

Lines 119 to 142 prompt the user to provide the names of the three attributes, the measure, and the saved filter included in the query defined on lines 144 through 170. **InputBox** objects, a standard Visual Basic for Applications object for requesting information from a user, store users' responses in the variables you declared earlier in the procedure. When prompted for the information, the user must precisely identify the attribute, measure, and filter values, as they appear in the list boxes in the **Define the Query** spreadsheet.

Lines 173 through 204 execute the query as before, substituting values provided by the user as arguments in the commands defining the query and the ensuing report.

Exercise 13: Slow Query Warning

In this exercise you include a section that evaluates the performance cost of a user-defined query, warning the user if the query accesses large tables.

```
Option Explicit 'All variables must be declared explicitly

 'MetaCube API Exercise 13: Slow Query Warning

 Sub MetaCube_API()

 'Declare Variables and Constants
     'Object Variables
     Dim MyMetabase As Object, _
         FilterFolder As Object, _
         MyQuery As Object, _
         MyMetaCube As Object, _
         MySummaryCategory As Object, _
         MySortCategory As Object, _
         MyPivotCategory As Object

     'Excel Variables
     Dim ReportRange As Range, _
         MyListBox As ListBox

     'Other Variables
     Dim MyData As Variant, _
         ArrayofItems As Variant, _
         DimensionCount As Integer, _
         MyFirstAttribute As String, _
         MySecondAttribute As String, _
         MyThirdAttribute As String, _
         MyMeasure As String, _
         MySavedFilter As String, _
        CostWarning As Integer

     Const OrientationColumn = 2
     Const SortDirectionDesc = 2
     Const SummaryTotal = 1
     Const DisplayStyleQuery = 2

 'Create & Define A Metabase: Log in to RDBMS, Open DSS System
     'Instantiate a Metabase object
     Set MyMetabase = CreateObject("Metabase")

     'Identify an ODBC Data Source
     Let MyMetabase.ConnectString = "Metademo"

```

```
44        'Specify a set of metadata
45        Let MyMetabase.Name = "MetaCube Demo"
46
47        'Specify login to database
48        Let MyMetabase.Login = "metademo"
49        'Passes value to database on connection
50
51        'Specify database password
52        Let MyMetabase.Password = "Metademo"
53
54        'Log in to demonstration database, open DSS System
55        MyMetabase.Connect
56
57    'Identify folder containing filters
58    Set FilterFolder = MyMetabase.RootFolder. _
59        Folders.Item("Public Filters")
60
61    'Query Interface:  Attributes
62        Worksheets.Item("Define the Query").Activate
63
64    'Cycle through DSS System's dimensions
65    For DimensionCount = 0 To _
66                MyMetabase.Dimensions.Count - 1
67
68        'Get array of attributes
69        Let ArrayofItems = _
70          MyMetabase.Dimensions.Item(DimensionCount). _
71          AttributeNames(DisplayStyleQuery).ArrayValues
72
73        'Create listboxes
74        Set MyListBox = ActiveSheet.ListBoxes.Add _
75        ((DimensionCount * 80), 50, 70, 100)
76       'A list box for each dimension, each further to the right
77
78        'Populates each list box w/ attributes of a dimension
79        MyListBox.AddItem ArrayofItems
80
81    Next DimensionCount
82
83    'Query Interface:  Measures
84
85    'Create one list box for measures
86    Set MyListBox = ActiveSheet.ListBoxes.Add _
87        ((DimensionCount * 80), 50, 70, 100)
88    'Previous "Next DimensionCount" added 1, moves listbox right
89
90    'Get array of available measure names
91    Let ArrayofItems = _
92        MyMetabase.FactTables. _
```

```
93        Item("Sales Transactions"). _
94        MeasureNames(DisplayStyleQuery).ArrayValues
95
96    'Populate listbox with array of measure names
97    MyListBox.AddItem ArrayofItems
98
99    'Query Interface:  Saved Filters
100
101   'Create listbox for saved filters
102   Set MyListBox = ActiveSheet.ListBoxes.Add _
103       (((DimensionCount + 1) * 80), 50, 120, 150)
104   'Increments "DimensionCount" by 1, moves listbox to right
105
106   'Get array of filters in root folder owned by METAPUB
107   Let ArrayofItems = _
108       FilterFolder.FilterNames _
109       ("METAPUB", "").ArrayValues
110
111   'Populate list box with array of filter names
112   MyListBox.AddItem ArrayofItems
113
114   'Retrieve Names of Attributes, Measure, Filter from User
115 DefineQuery:
116   Let MyFirstAttribute = _
117           InputBox(Prompt:= _
118           "Enter an attribute on which to query:", _
119           Title:="First Attribute")
120
121   Let MySecondAttribute = _
122           InputBox(Prompt:= _
123           "Enter the name of another attribute:", _
124           Title:="Second Attribute")
125
126   Let MyThirdAttribute = _
127           InputBox(Prompt:= _
128           "Enter the name of a third attribute:", _
129           Title:="Attribute Organized by Columns")
130
131   Let MyMeasure = _
132           InputBox(Prompt:= _
133           "Enter a measure on which to query:", _
134           Title:="Measure")
135
136   Let MySavedFilter = _
137           InputBox(Prompt:= _
138           "Enter the name of a saved filter:", _
139           Title:="Filter")
140
141   'Define the Query
```

```
142  Set MyQuery = MyMetabase.Queries.Add("A New Query")
143  'Adds query to MyMetabase's collection of queries
144
145  Set MySummaryCategory = _
146      MyQuery.QueryCategories.Add(MyFirstAttribute)
147
148  Set MySortCategory = MyQuery.QueryCategories.Add _
149      (MySecondAttribute)
150  'Sort the values of this attribute in reverse-alphabetical
    order
151  Let MySortCategory.SortDirection = SortDirectionDesc
152
153  Set MyPivotCategory = _
154      MyQuery.QueryCategories.Add(MyThirdAttribute)
155  'Pivot this attribute to the column orientation
156  Let MyPivotCategory.Orientation = OrientationColumn
157
158  'Add measure to MyQuery's collection of measures
159  MyQuery.QueryItems.Add MyMeasure
160
161  'Add second measure, on which to perform calculation
162  MyQuery.QueryItems.Add _
163      "Abs_Change(" + MyMeasure + ")"
164
165  'Add saved filter; specifies filter name, who saved, and
    folder
166  MyQuery.Filters.AddSaved _
167      MySavedFilter, "METAPUB", FilterFolder
168
169  'Display Query Definition
170  MsgBox MyQuery.SQL
171  'SQL property shows SQL generated for query before execution
172
173  'Check the cost of tables to be accessed by MetaCube, warn user
174  If MyQuery.Cost > 300 Then    'Arbitrary value
175
176      CostWarning = MsgBox _
177          (Prompt:= _
178      "This query may be slow.  Run query anyway?", _
179          Title:="Query Cost Warning", _
180          Buttons:= _
181          vbYesNo + vbExclamation + vbDefaultButton2)
182
183      'Warn user
184      If CostWarning = vbNo Then
185          GoTo DefineQuery:          'Allow user to redefine query
186      End If                'Otherwise, execute query
187
188  End If
```

```
189
190   'Get Query Results, Define Report
191
192   'Add cube to MyQuery's cube collection
193   Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
194
195   'Subtotal: Regional sales for each brand
196   MyMetaCube.Summaries.Add _
197       MySummaryCategory, SummaryTotal
198
199   'Format data as an array VB can display, store in variable
200   Let MyData = MyMetaCube.ToVBArray
201   'The ToVBArray method implicitly requires MetaCube
202   'to execute the query on the relational database
203
204   'Clear "Query Report" Worksheet
205   Sheets("Query Report").Activate
206   Cells.Select
207   Selection.ClearContents
208
209   'Excel Code:  Defines Range of Cells, Presents Data in
    Worksheet
210   Worksheets.Item("Query Report").Activate
211   Set ReportRange = _
212       ActiveSheet.Range _
213       (ActiveSheet.Cells(1, 1), _
214       ActiveSheet.Cells _
215       (MyMetaCube.Rows, MyMetaCube.Columns))
216   Let ReportRange.Value = MyData
217   ReportRange.EntireColumn.AutoFit  'Sizes columns
218
219   End Sub
```

Explanation of Exercise 13

Allowing a user to define a query introduces the risk of poorly designed queries that access extremely large tables. When optimizing the SQL for a particular query, MetaCube assesses the relative performance costs of accessing different tables. These costs, which typically correspond to the number of rows in each table, are assigned by the administrator, who generates a metadata description for each fact or summary table.

When generating the SQL for a query, MetaCube identifies the table that features the lowest possible cost and involves a minimal number of joins. The cost of the fact or summary table for which MetaCube generates SQL represents the cost of the query itself, regardless of the number of rows that the query selects from that table.

Each fully defined Query object has a **Cost** property, the value of which is calculated by the MetaCube analysis engine when it generates SQL for that query. On the basis of this value's magnitude, this exercise can warn the user of a possible delay before the query executes and offer the option of redefining the query.

Line 174 evaluates the cost of the query represented by the **MyQuery** object, setting a threshold of 300, which, if exceeded, prompts MetaCube to warn the user. If the query's cost does not exceed this value, the application's execution path ignores the rest of this section of code. In the event a warning is warranted, the application displays the warning in a **MsgBox** object, as specified in lines 176 through 181. The threshold of 300 is artificially low to ensure that the slow query warning displays for this example.

Unlike the message box created in line 170 to show the SQL generated for a query, this message box returns a user-defined value to the **CostWarning** integer. This value depends on the button clicked by the user, with each button corresponding to a numeric argument. If the user clicks the **No** button, indicating that he or she does not want to execute the query as currently defined, the nested condition in line 184 is satisfied and line 185 executes. This line returns the user to the section beginning on line 115, identified in this exercise by the header `DefineQuery`. By returning to this section, the application forces the user to redefine the query. If the user clicks the **Yes** button, the query runs anyway, ignoring the nested code.

Chapter

3

The Metabase Class of Objects

In This Chapter

This chapter introduces the **Metabase** class of objects, which represent a virtual multidimensional database.

The Metabase Class of Objects

Before you can build a query, you must define a logical representation of a multidimensional database by creating an instance of the **Metabase** class of objects. Each Metabase object is the logical representation of a different multi-dimensional database, and the properties of a particular Metabase object specify the characteristics of that multidimensional database, including the relational database on which your multidimensional system is based, the multidimensional interface through which it will be accessed, and the analytical functions available.

For every MetaCube procedure, a Metabase object is the founding or "master" object, with all other objects existing as children, grandchildren, great-grandchildren, and so on, of this master object.

As a class, Metabase objects stand at the top of a hierarchy of MetaCube object classes. A Metabase object is the parent of a *collection* of queries accessing data from the tables represented by that Metabase object. Objects of the same class are organized into a collection belonging to the parent object. A metabase's collection of Query objects represents the different ways in which that instance of a multidimensional database can be queried. A query cannot exist but within a collection belonging to the Metabase object; in fact, a free-standing query would be meaningless since a query is defined in terms of the multidimensional view of the database (that is, instance of a Metabase object) that it queries.

All but the most low-level objects are parents of one or more collections. For example, each query is the parent of a collection of selected attributes (**QueryCategories**), measures (**QueryItems**), filters (**Filters**), and reports (**MetaCubes**). In turn, attributes, measures, filters, and reports define different aspects of a query. More generally, each collection defines an aspect of the parent object.

At the outset of any MetaCube procedure, you must create an instance of the **Metabase** class. In Visual Basic and Visual Basic for Applications, the **CreateObject** function performs this task. The class of the object you want to create is an argument in the **CreateObject** statement. Before you instantiate a Metabase object, you must declare an object variable and set that variable equal to the newly created instance of the **Metabase** class of objects:

```
Dim MyMetabase as Object
Set MyMetabase = CreateObject("Metabase")
```

The environment in which you develop MetaCube applications recognizes the **Metabase** class of objects because MetaCube registers its library of objects when you install MetaCube. While the object variable representing a particular instance of a **Metabase** class of objects can have any unique name, the **Metabase** class name that you pass to the **CreateObject** function is always the same.

Other object classes are not instantiated by a **CreateObject** function or that function's analog in other development environments. Instances of object classes other than the Metabase class exist within collections directly or ultimately belonging to an instance of a Metabase object. To instantiate a Query object for example, you need only add a query to a Metabase object's collection of queries:

```
MyMetabase.Queries.Add "New Brand/Region Query"
```

or

```
Dim MyQuery as Object
Set MyQuery = MyMetabase.Queries.Add _
    ("New Brand/Region Query")
```

Metabase Properties

Many of the properties of the Metabase object correspond to the preferences of MetaCube Explorer or Warehouse Manager. These properties specify the characteristics of the virtual multidimensional database that the Metabase object represents. Figure 3-1 summarizes the properties of the **Metabase** class of objects.

Connection Information in MetaCube 4.0

MetaCube 4.0 introduces two major new features: Web Explorer, a data access tool that can be used from an Internet browser, and Secure Warehouse, a utility for controlling user access to data warehouses. To incorporate these new features, MetaCube now has the capability of handling connection information for a user differently than it did in previous versions.

In MetaCube 4.0, user information can be stored in two places: the registry of the computer where the MetaCube engine is running and the metadata **Client** table for the current database connection. The user properties stored in the registry provide default connection information. They allow MetaCube Web Explorer users to connect to the database via MetaCube. All other user properties are stored in the **Client** table in the database.

When Secure Warehouse is used to create users, entries for those users are created in the registry of the computer running the MetaCube analysis engine to which Secure Warehouse is connected. Typically that computer is a middle-tier NT server. If there are multiple middle-tier NT servers, entries must be created in the registry of each machine for the users that connect through the MetaCube analysis engine running on that computer.

Users of client/server applications, such as MetaCube Explorer, do not need registry entries to enable a database connection. These applications always set their connection properties explicitly, either through user input or by using the values defined in the client's **MetaCube.ini** file.

Figure 3-1
Metabase Class of Objects: Properties

Property	Description/Example
Application	Object. This property returns the application object, that is, the MetaCube analysis engine. This property, though essentially useless, is required by OLE. `MsgBox MyMetabase.Application.Version.`
ClientType	A read-only property that identifies the development environment in which the Metabase object has been instantiated; determines the format in which MetaCube returns strings. See "Related Constants" on page 3-14 for more information on constants. `MsgBox MyMetabase.ClientType`
Configuration	Identifies a set of properties stored as parameters in **MetaCube.ini.** String. In version 4.0 and later releases of MetaCube, the MetaCube analysis engine engine does not use this property. It is retained, however, for compatibility with previous versions. `MyMetabase.Configuration = "Default"`
ConnectDatabase	Identifies the vendor of the RDBMS. See "Related Constants" on page 3-14 for more information on vendor constants. If not specified explicitly, this property defaults to `Informix`. `MyMetabase.ConnectDatabase = DBVendorInformix`

Connected	Indicates whether a Metabase object has connected to RDBMS. Returns true if connected.
	`If MyMetabase.Connected = True _`
	`Then MsgBox "Hooray!"`

ConnectString	Identifies an ODBC data source. To establish a database connection, this property must be set explicitly. `MyMetabase.ConnectString = "MetaDemo"`
CurrentTime	Retrieves the current date and time from the PC clock. Use with QueryBack. Variant (date). `MsgBox MyMetabase.CurrentTime`

(1 of 6)

Property	Description/Example
DatabaseDBSpaces	ValueList of all valid dbspaces for the current database connection. The contents of this ValueList are used to make a list of possible dbspaces, from which one can be chosen to store a user's QueryBackJob objects. `MsgBox MyMetabase.DatabaseDBSpaces`
DatabaseRoles	ValueList of all Informix roles defined in metadata. The contents of this ValueList are used to make a list of possible database roles that can be assigned to users managed in MetaCube Secure Warehouse. `MsgBox MyMetabase.DatabaseRoles`
DatabaseUsers	ValueList of all Informix users defined in the database. The contents of this ValueList are used to make a list of possible users, from which users that will be managed in Secure Warehouse are selected. `MsgBox MyMetabase.DatabaseUsers`
DataSkip	This long value property determines whether an Informix RDBMS can skip locked or otherwise unavailable rows when attempting to retrieve data. See constants below. MetaCube enables or disables data skip, as specified, upon connection to an Informix RDBMS. Defaults to `2`, the constant for accepting DataSkip default. `MyMetabase.DataSkip = DataSkipOff` For more information, see the DATASKIP entry in the *Informix Guide to SQL: Syntax*.
DataSources	ValueList of ODBC data sources available to MetaCube. The contents of this ValueList are used to make a list of database connections from which a user can select one. `MsgBox MyMetabase.DataSources`

(2 of 6)

Property	Description/Example
Explain	Setting this Boolean property to true prompts an Informix RDBMS to record information such as the execution plan and the cost of each query, as generated by the database-server optimizer, in a server-side file titled **sqexplain.out**. The default value of this property is false. MetaCube activates this database feature, if requested, upon connection to an Informix RDBMS. `MyMetabase.Explain = True` See the SET EXPLAIN entry in the *Informix Guide to SQL: Syntax* for more information.
LastUpdate	Stores the date on which the DSS System's metadata was last modified. Variant (date). `MsgBox MyMetabase.LastUpdate`
LocalMetamodelFile	Name of the file storing the local copy of metadata. Default read from **MetaCube.ini file**. String. `MyMetabase.LocalMetamodelFile = "MetaCube.DSS"`
Login	User name passed to RDBMS for login. Default read from **MetaCube.ini**. String. `MyMetabase.Login = "MetaDemo"`
MaxTotalFetches	Specifies the number of rows MetaCube retrieves before aborting a query. `MyMetabase.MaxTotalFetches = 4000`
MetamodelNames	ValueList of all DSS System names in metadata. With MetaCube 4.0, this property has been replaced by the **Names** property of the DSSSystems collection (that is, **Metabase.DSSSystems.Names**). `MsgBox MyMetabase.MetamodelNames`
MetaSchema	Identifies a schema (or prefix) for metadata tables. To establish a database connection, this property must be set explicitly. `MyMetabase.Schema = "MetaCube."`

(3 of 6)

Property	Description/Example
Name	Default property. The name of the DSS System to be created or the name of the DSS System currently open. String. `MyMetabase.Name = "MetaCube Demo"`
Optimization	This Boolean property determines the extent to which the optimizer of an Informix RDBMS evaluates every possible execution strategy for an SQL query. The default value of true allows the RDBMS optimizer time to consider all reasonable join strategies and indexes. Setting this property to false reduces the time devoted to optimizing SQL queries but might preclude the RDBMS optimizer from deploying the most efficient execution plan. MetaCube sets the optimization level upon connection to an Informix RDBMS. `MyMetabase.Optimization = False` See the SET OPTIMIZATION entry in the *Informix Guide to SQL: Syntax* for more information.
Parent	Returns the application object. Required by OLE for multi-threading. `MsgBox MyMetabase.Parent`
Password	Password submitted to the database server. Setting this property through the programming interface does not automatically record a value in MetaCube's initialization file. String. `MyMetabase.Password = "MetaDemo"`

(4 of 6)

Property	Description/Example
PDQPriority	This property designates the parallel data query (PDQ) priority of all decision support system queries submitted by MetaCube to the Informix database. PDQ priorities, which can range from 0 to 100, determine the extent to which the Informix database executes queries in parallel. A value of 0 explicitly precludes any parallel operations, a value of 1 enables only parallel scans, and values between 2 and 100 represent the percent of available system resources that queries against this decision support system can consume. In multiprocessor systems, high PDQ priorities enable the database to process queries faster. PDQ priorities can be limited by environment variables and files established by the database administrator when configuring the Informix RDBMS. The long value of this property defaults to -1, indicating that MetaCube will not set PDQ priorities. Accept this default when connecting to databases that do not support PDQPriority, such as Microsoft Access and Informix Standard Engine, or when submitting queries to a uniprocessor server. If any value other than the default is specified for this property, MetaCube attempts to set PDQPriority for all queries upon connection to an Informix RDBMS. `MyMetabase.PDQPriority = 50` For more information about PDQPriority, see the *Administrator's Guide for Informix Dynamic Server*.
PublicUser	Read-only string. Identifies the user empowered to save queries and filters that other users can access in MetaCube Explorer and MetaCube for Excel. This string is set to **metapub** for current releases of Explorer and MetaCube for Excel. In future releases, this property may be variable, in which case applications should use this property to identify the public user. `MsgBox MyMetabase.PublicUser`
Role	A string containing the database role of the currently connected user. If a user is not connected, the string is empty. `MyMetabase.Role = "Analyst"`

(5 of 6)

Property	Description/Example
SuppressDialogs	Boolean. A true value precludes the MetaCube Status window from appearing in your application. Defaults to true. `MyMetabase.SuppressDialogs = True`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a Metabase object is one of the following: ■ Completely valid ■ Invalid because an object in the collections owned directly or indirectly by the Metabase object is invalid ■ Invalid because the Metabase object itself is invalid If you have not invoked the **Verify** method, this property defaults to `VerifiedNever`, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyMetabase.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the Metabase object's metadata. This will not include errors in the metadata for objects owned by the Metabase object. `MsgBox MyMetabase.VerifyResults.TabbedValues`

(6 of 6)

Metabase Methods

Metabase properties allow you to define or retrieve different characteristics of the Metabase object. **Metabase** methods allow you to perform different operations on a metabase, such as logging in, creating a new DSS System, or opening a query. You should already be familiar with the most common **Metabase** method, the **Connect** method. The functionality of **Metabase** methods falls into three categories:

- Connecting and disconnecting to the relational database and opening and closing DSS Systems

- Creating and deleting DSS Systems and subsequently updating the metadata
- Saving queries and opening or deleting saved queries

Figure 3-2 summarizes the **Metabase** object class's methods.

Figure 3-2
Metabase Class of Objects: Methods

Method	Description/Example
CloseDSSSystem	Closes the current DSS System. `MyMetabase.CloseDSSSystem`
Connect	Logs in to the RDBMS; opens a DSS System and implicitly calls the **OpenDSSSystem** method. `MyMetabase.Connect`
CreateNew	Creates a new DSS System. The name of the DSS System is specified by Metabase's **Name** property. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to true) can call this method. `MyMetabase.CreateNew`
DBLogin	Logs in to the RDBMS; does not open a DSS System. `MyMetabase.DBLogin`
DBLogout	Disconnects MetaCube from the RDBMS. `MyMetabase.DBLogout`
DeleteMetamodel	Deletes the DSS System from metadata in the RDBMS. Arguments: DSS System name. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to true) can call this method. This method immediately deletes a DSS System. You do not have to call **Metabase.Save** for this method to take effect. Any references to the deleted DSS System as the default DSS System are set to null. `MyMetabase.DeleteMetamodel  "Demo"`

(1 of 3)

Method	Description/Example

DeleteQuery	Deletes from the RDBMS's metadata the definition of a query, as saved by the **SaveAs** method of the **Queries** class of objects. Arguments: the query's name, as a string; the query's author, as a string; the folder into which the query has been saved, as an object.
	`MyMetabase.DeleteQuery "Saved Query", _`
	`"MetaDemo",FolderObject`

Disconnect	Closes the DSS System. Disconnects from the RDBMS. `MyMetabase.Disconnect`
DoSQL	Executes a non-SELECT SQL statement. `MyMetabase.DoSQL "CREATE TABLE . . ."`
OpenDSSSystem	Opens the DSS System specified by **Metabase.Name**. This method is equivalent to **OpenDSSSystem2** when that method is set to True. `MyMetabase.OpenDSSSystem`

OpenQuery	Returns an instance of a Query object bearing the definition of a saved query but none of the query's associated data, reports, or charts. Arguments: the query's name, as a string; the query's author, as a string; and the folder into which the query has been saved, as an object.
	`Set MyQuery = MyMetabase.OpenQuery _`
	`("Saved Query", "MetaDemo", _`
	`MyMetabase.RootFolder)`

OpenQueryStorage	Returns a structured storage object representing a Query object and its result, as saved by the **SaveStorage** method of the **Queries** class of objects. This method, which is actually beyond the OLE paradigm, can only be used in C++ and other development environments that directly support the COM interface.

(2 of 3)

Method	Description/Example
RemoteConnect	Using a user's connection information stored in the registry, this method connects a remote client, such as a MetaCube Web Explorer, to the RDBMS and opens a DSS System. Arguments: the user's name, as a string; the user's password, as a string; and optionally, a DSS System name, as a string. If no DSS System name is supplied, MetaCube uses the user's default DSS System. `MyMetabase.RemoteConnect "WebUser", _ "Password", "Demo"`
Save	Saves changes in the DSS System to the RDBMS metadata. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to True) can call this method. `MyMetabase.Save`
SaveAs	Saves all metadata in the RDBMS describing a DSS System, including folders, filters and queries, under a new name. Arguments: New DSS System name. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to true) can call this method. When you use **SaveAs** to create a new DSS, remember that initially no users have access to the new DSS System. The data warehouse administrator must give users access to the new DSS System, or you must use the programming interface to grant access to users. `MyMetabase.SaveAs "New Name"`
Verify	Reviews the metadata definitions represented by the Metabase objects and other objects within its collections or subcollections, returning values to the Metabase object's **Verified** and **VerifyResults** properties. `MyMetabase.Verify`

(3 of 3)

Related Constants

Figure 3-3 summarizes the numeric arguments for the **ClientType** property, including the names of constants declared in the **MetaCons.bas** file.

Figure 3-3
Client Type Constants

Development Environment	MetaCons.bas Constant Name	Constant
Visual Basic	ClientTypeVB	1
Visual Basic for Applications	ClientTypeExcel	2
C	ClientTypeC	3

Figure 3-4 summarizes the numeric arguments for the **ConnectDatabase** property, including the names of the constants declared in the **MetaCons.bas** file.

Figure 3-4
Database Vendor Constants

Database Vendor	MetaCons.bas Constant Name	Constant
Microsoft Access	DBVendorAccess	2
Informix Online	DBVendorInformix	5

Figure 3-5 summarizes the numeric constants to be specified when setting the **DataSkip** property.

Figure 3-5
Data Skip Constants

Data Skip Functionality	MetaCons.bas Constant Name	Constant
Abort query if any requested record is unavailable.	DataSkipOff	0
Skip any unavailable records, returning what values are available.	DataSkipOn	1
Accept database default for Data Skip option.	DataSkipDefault	2

Figure 3-6 below shows the numeric arguments returned by the **Verify** method.

Figure 3-6
Verify Codes

Verification Status	MetaCons.bas Constant Name	Constant
Object has never been verified	VerifiedNever	0
Object verified successfully	VerifiedGood	1
Object itself verifies, but other objects in its collections or subcollections do not	VerifiedBadBelow	2
Object itself fails to verify	VerifiedBad	3

Metabase Collections

Figure 1-2 on page 1-9 offers a simplified view of MetaCube's hierarchy of object classes, representing a collection of **Queries** as the only collection directly descended from the Metabase object. For the purposes of designing the simple query application featured in the preliminary tutorial, a detailed understanding of every collection spawned by the **Metabase** class of objects was unnecessary. Actually, ten collections belong to the Metabase object, as shown in Figure 1-3 on page 1-10 and described in Figure 3-7.

Figure 3-7
Metabase Class of Objects: Collections

Collection	Description/Example of Add Method

Dimensions	Consists of all dimensions in a DSS System. A subset of this collection belongs to the **FactTables** class of objects. If an item exists in both collections, deleting an item from this collection also deletes the item from the FactTable objects' collections. To instantiate a Dimension object, you must specify the name of the dimension, its type, the name of the database schema owning the underlying table, the name of the table itself, the name of the column joining that table to the fact table, and the name of the column storing aggregate level information.
	`MyMetabase.Dimensions.Add`
	`  "Time", DimensionTypeTime, "MetaDemo", _`
	`  "TIME_DIMENSION", "DAY_CODE",  "AGG_LEVEL"`
	See "The Dimensions Collection's Add Method" on page 4-4 for more details.

DSSSystems	Consists of all DSSSystem objects in metadata. These objects are loaded after the user connects to the database. This collection does not feature an **Add** method, because MetaCube generates the items within this collection by reviewing the metadata tables. See "The DSSSystems Class of Objects" on page 10-3 for more details.

(1 of 3)

Collection	Description/Example of Add Method

Extensions	Consists of libraries of functions, developed as extensions to MetaCube's analytical engine. Consultants, third-party vendors, and sophisticated customers can develop extensions in C++ from a template provided with MetaCube's analysis engine. The syntax for developing extensions is unrelated to the programming interface of MetaCube's hierarchy of object classes. The code is subsequently compiled in a separate file, which functions like a dynamic link library. Each extension may consist of one or many functions. Any time an application refers to an individual function, the entire extension is invoked. Once developed, extensions are registered through MetaCube's programming interface by instantiating an Extension object, with an argument that identifies the file containing compiled code.
	`MyMetabase.Extensions.Add _`
	`"c:\metacube\mcplgmn.mcx"`
	When instantiating an Extension object, include the name of the extension in MetaCube's initialization file, **metacube.ini**, enabling the extension whenever MetaCube launches.

FactTables	Consists of all fact tables in this DSS System. To instantiate a FactTable object you must specify the name of the object, the name of the database schema owning the underlying table, and the name of the database table itself that represents the fact table.
	`MyMetabase.FactTables.Add "Sales Transactions", _`
	`"METADEMO", "SALES_TRANSACTIONS"`
	See "The FactTables Class of Objects" on page 6-3 for more details.

Queries	Consists of any saved queries that have been opened during a session. Ad hoc queries are also instantiated and defined within this collection, as shown in "Exercise 2: Defining A Query" on page 2-8. To instantiate a Query object, specify the name of the new object as an argument to the **Add** method. `MyMetabase.Queries.Add "My New Query"` See "The Queries Class of Objects" on page 8-3 for more details.

(2 of 3)

Collection	Description/Example of Add Method
QueryBackJobs	Consists of all QueryBack jobs for a DSS System that were submitted by this user, both pending and complete. The collection of QueryBackJob objects does not feature a formal **Add** method because the **Submit** method of the Query object actually creates a QueryBack job. See "The QueryBackJobs Class of Objects" on page 8-78 for more details.
RootFolder	This is a special type of collection, consisting of only one object, which itself can be the parent of other objects of a similar type. This object is the highest level object representing the storage interface queries and filters saved in MetaCube's metadata. See "The Folders Class of Objects" on page 7-3. This collection does not feature an **Add** method because any DSS System can only have one **RootFolder** object, which exists by default.
Schemas	Consists of all physical database schemas or table owners within the RDBMS and is useful for verifying metadata. This collection does not feature an **Add** method because MetaCube generates the items within this collection by reviewing the database's system tables. See "Schemas, Tables, Columns" on page 9-3 for more details.

SystemMessages	Consists of all system messages for a DSS System. To instantiate a SystemMessage object, deploy the **Add** method, specifying the text of the message and the date and time of its creation.
	`MyMetabase.SystemMessages.Add "Data Loaded", _`
	`MyMetabase.CurrentTime`
	See "The SystemMessages Class of Objects" on page 11-3 for more details.

Users	Consists of User objects representing MetaCube users. To instantiate a new User object, deploy the **Add** method for the Users collection, specifying the name of the new user. `MyMetabase.Users.Add "New User"` See "The Users Class of Objects" on page 10-4 for more details.

(3 of 3)

The Metabase object's collections encompass the entire range of MetaCube functionality. The Dimensions and FactTables collections represent a DSS System's metadata, which describes the data warehouse in natural business terms and configures the MetaCube engine to your data model.

Applications such as Warehouse Manager define the properties and invoke the methods of the objects contained within the Dimensions and FactTables collections to generate MetaCube's metadata. Applications such as Explorer review the properties of these objects to display the available attributes and measures on which to query and to filter. The SQL that the MetaCube analysis engine generates depends on the descriptions of fact tables, aggregate tables, dimension tables and attribute tables represented by these collections.

Objects in the FactTables and Dimensions collections thus lay the foundation for your data warehouse to process multidimensional queries. Objects in the Filters, Queries, and QueryBackJobs collections actually define those queries in the multidimensional terms inscribed by objects in the Dimensions and FactTables collections.

When you instantiate a QueryCategory object, for example, you must include an argument identifying the name of an Attribute object. The QueryCategory object belongs to a collection defining the attributes for a particular Query object. The Attribute object belongs to a collection owned by a Dimension object generally defining the available attributes for any query within a particular DSS System.

The **RootFolder** object class owns hierarchical collections of folders and subfolders by which the storage of Filter and Query objects is organized.

Objects in the Schema collection and their descendants represent a data dictionary of physical schemas, tables, and columns in the relational database. Objects in the Schema collection do not, however, store any of the corresponding multidimensional properties of these database structures. As such, this collection exists purely as a reference for supplying values to the properties of objects in the FactTables and Dimensions collection, which map the relationship between physical structures and multidimensional terms and logic.

The SystemMessages collection of objects store string information for distributing information to MetaCube users, such as the status of the database or the maintenance schedule.

The User and DSSSystem collections allow you to manage access to data the same as an administrator would do using MetaCube Secure Warehouse. You can control data by identifying which users can access DSS Systems, assigning mandatory filters to those users to limit what data they can see within a DSS System, and restricting access to QueryBack jobs so users can query data only at certain times.

Chapter

4

The Dimensions Class of Objects and Related Collections

In This Chapter

This chapter introduces the **Dimensions** class of objects and all the collections directly or indirectly belonging to objects of the **Dimensions** class. The **Dimensions** class and the collections belonging directly or indirectly to it allow you to develop procedures that create, edit, or access MetaCube's metadata.

The Dimensions Class of Objects

A Dimension object represents a set of hierarchically related categories for grouping transactional data. For example, a Dimension object could represent a Time dimension, which stores the relationships between days, weeks, months, and years.

In Explorer, each dimension appears in a separate list box, which displays that dimension's attributes. An icon, a default filter, and a set of attribute names are associated with that dimension

Each category, or level in the dimensional hierarchy, corresponds to a DimensionElement object. MetaCube navigates from one level in the hierarchy to another by joining columns represented by a DimensionElement object. Such columns typically identify each value within that hierarchy level by a unique numeric code. Because the user typically never views these codes, only applications performing internal functions on MetaCube's metadata will deploy DimensionElement objects.

Instead, users define the groupings for their queries by attributes, represented by the Attribute object, which store descriptive terms associated with a particular DimensionElement object. This section discusses both dimension elements and attributes as collections that belong to a particular dimension.

The Dimensions Collection's Add Method

The **Add** method for the Dimensions collection requires six arguments, which you must enclose in parentheses to return the instance of the Dimension object to an object variable.

```
MyMetabase.Dimensions.Add Name, DimensionType, Schema, _
    Table, Column, AggLevelColumn
```

where **MyMetabase** is the instantiation of the Metabase object. The following example creates an object for the Time dimension:

```
Set MyDimension = MyMetabase.Dimensions.Add _
    ("Time", DimensionTypeTime, "MetaDemo", _
    "TIME_DIMENSION", "DAY_CODE",  "AGG_LEVEL")
```

Dimension Properties

The properties of the **Dimension** object itself largely correspond to the fields in MetaCube Warehouse Manager's **Dimension** tab.

Figure 4-1
Dimensions Class of Objects: Properties

Property	Description/Example
AggLevel-Column	String. The dimension table column storing aggregate level identifiers, which indicate the row in the dimension table where summary tables can join to find unique values for each dimension element. This property can be specified or left empty (NULL): ■ When empty, the **DimensionElement.AggLevel** property is also NULL and **DimensionElement DET** values must be set. ■ When specified, the **DimensionElement.AggLevel** property may either be NULL or contain an aggregate level value. `MyDim.AggLevelColumn = "Agg_Level"`
Attribute-Names	ValueList of names for all of the dimension's attributes. Arguments: Display type constants, to display items validated for queries, filters, or both. See below. `MsgBox MyDim.AttributeNames (2).TabbedValues`
BaseElement	Object. The dimension element that represents the lowest level of detail in the dimensional hierarchy. Read-only. `MsgBox MyDim.BaseElement.Name`
Column	String. The dimension table column joining that table to the fact table. This column also typically corresponds to the base dimension element. `MyDim.Column = "DAY_CODE"`
Current-PeriodColumn	String. The column in the dimension table storing the current period flag. Null for nontime dimensions. `MyDim.CurrentPeriodColumn = "CURRENT"`
DefaultFilter	Object. The saved filter for this dimension identified as the default, if a default exists. `MsgBox MyDim.DefaultFilter.Name`
Dimension-Type	Integer. Indicates if a dimension stores time relationships. Time dimensions have different properties and requirements than other dimensions. Default: **DimensionTypeNonTime**. See tables of constants. `MyDim.DimensionType = DimensionTypeTime`

(1 of 3)

Property	Description/Example
IconBitmap	String. The icon associated with a dimension is converted from a bitmap to string information, and stored in the database. This value's property is thus the string representation of the icon. `MsgBox MyDim.IconBitmap`
IconName	String. Name of original bitmap file for storing icon that represents this dimension. `MyDim.IconName = "TIME.ICO"`
Name	String. Name of the dimension. Default property. `MyDim.Name = "Time"`
NoFilterLabel	String. A label indicating that no filters have been applied to the parent dimension. No default. `MyDim.NoFilterLabel = "All Products"`
Parent	Object. The Metabase object. `MsgBox MyDim.Parent`
Schema	String. The physical location of the dimension table. `MyDim.Schema = "MetaDemo"`

(2 of 3)

Property	Description/Example
Table	String. The name of the dimension table. `MyDim.Table = "TIME_DIM"`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a Dimension object is one of the following: ■ Completely valid ■ Invalid because an attribute or dimension element object belonging to the Dimension object is invalid ■ Invalid because the Dimension object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyDimension.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the Dimension object's metadata. This does not include errors in the metadata for objects owned by the Dimension object. `MsgBox MyDimension.VerifyResults.TabbedValues`

(3 of 3)

Dimensions Methods

Aside from the **Verify** method, which reviews the validity of the Dimension object's metadata properties, the **Dimensions** class of objects has only one method, the **WriteIcon** method, which creates an icon from a value stored in the database. Warehouse Manager originally generates the string value from an icon file in order to store the icon's image in the database. Warehouse Manager accomplishes this task by assigning values to two Dimension object properties:

```
MyDimension.IconName = "TIME.ICO"
MyDimension.IconBitMap = "AF50..."
```

The **WriteIcon** method converts this string value, represented by the Dimension object's **IconBitmap** property, back to an icon file, as recognized by the **IconName** property. When invoking this method, you must specify as an argument the directory in which you want the icon file created, as in the following example:

```
MyDimension.WriteIcon "C:\METACUBE"
```

Related Constants

Figure 4-2 summarizes the numeric arguments for the **AttributeNames** property, including the names of constants declared in the **MetaCons.bas** file. The **MeasureNames** property of the FactTable object, which also requires a numeric argument specifying the types of items to display, can use the same constants explained below.

Figure 4-2
Display Type Constants for Attributes and Measures

Property	MetaCons.bas Constant Name	Constant
Display neither	DisplayStyleNone	0
Display items valid for filtering	DisplayStyleFilter	1
Display items valid for querying	DisplayStyleQuery	2
Display items valid for filtering and querying	DisplayStyleBoth	3

Figure 4-3 summarizes the numeric argument for the **DimensionType** property, including the names of constants declared in the **MetaCons.bas** file.

Figure 4-3
Dimension Type Constants

Type of Dimension	MetaCons.bas Constant Name	Constant
Time dimension	DimensionTypeTime	1
Any other type of dimension	DimensionTypeNonTime	0

Dimensions Collections

After you define the general characteristics of a dimension, you can specify in greater detail each of the components of a dimension. They are organized into two collections: the dimension elements that define the hierarchy levels within a dimension and the attributes that in turn describe each level. Figure 4-4 summarizes these collections:

Figure 4-4
Dimensions Class of Objects: Collections

Collection	Description
Attributes	Consists of the Attribute objects describing every dimension element within this dimension. Each DimensionElement object individually owns a subset of this collection, containing the Attribute objects corresponding to that dimension element.
Dimension-Elements	Consists of a set of hierarchically related DimensionElement objects, each representing a column in the dimension table.

Each of these collections is discussed in the following pages, beginning with the collection of DimensionElement objects, because Attribute objects belong to collections owned both by Dimension and by DimensionElement objects.

The DimensionElements Class of Objects

A DimensionElement object identifies the column in the relational database defining a particular level in a dimensional hierarchy. This column typically stores a unique code for each value at that level, and can join to fact or summary tables to enable MetaCube to consolidate or group data by higher levels of summarization in the dimension table. By drilling down or up on attribute values associated with a particular dimension element, users can easily navigate through a dimensional hierarchy.

Since most users query on the descriptive terms represented by Attribute objects, front-end applications such as Explorer might never instantiate this object.

The DimensionElements Collection's Add Method

The **Add** method for instantiating a DimensionElement object requires three arguments that correspond to several properties described in the next section. In order of appearance, these arguments are the dimension element's name, the column storing the actual values of the dimension element, which also joins the dimension element to a separate attribute table if it exists, and the identifier within the **Aggregate Level** column flagging the rows in the dimension element column that store only distinct values of the dimension element:

```
MyDimension.DimensionElements.Add Name,
DimensionToAttributeColumn, AggLevel
```

For example, to add a dimension element named **Brand**, stored in a column named **BRAND_CODE** with an aggregate level value of 4, we would enter the following command:

```
MyDimension.DimensionElements.Add "Brand", "BRAND_CODE", 4
```

AddWithDET Method

The **DimensionElement.AddWithDE**T method is different from the **DimensionElement.Add** method for instantiating a DimensionElement object. The **DimensionElement.Add** method issued to instantiate a DimensionElement object that identifies the column in the relational database that defines a particular level in a dimensional hierarchy.

The **DimensionElement.AddWithDET** method is used for instantiating a DimensionElement object with dimension element tables. This type of DimensionElement object has no aggregate level.

The **DimensionElement.AddWithDET** method for instantiating a DimensionElement object with DETs takes six parameters:

- Name of the dimension element
- Name of the column storing the actual values of the dimension element
- Name of the schema/owner of the dimension element table
- Name of the dimension element table
- SQL statement that creates and populates the dimension element table

- Options for the CREATE TABLE statement

The **AddWithDET** declaration is as follows:

```
AddWithDET(Name As String, DimensionToAttribute As String,
DETSchemaName As String, DETName As String, DETCreatestmt As
String, DETCreateOptions As String)
```

For example, the following command adds a dimension element with DETs named **DET_Brand**:

```
MyDimElem.AddWithDET "Brand","Brand_Code","metacube","DET_Brand", _
SQLString,"IN MyDBSpace EXTENT SIZE 16"
```

SQLString is a defined variable containing the SQL statements required to create the DET:

```
SELECT DISTINCT
    metademo.product_dimension.brand_code
    metademo.product_dimension.company_code
FROM
    metademo.product_dimension
WHERE
    metademo.product_dimension.brand_code IS NOT NULL
```

DimensionElements Properties

Many of a DimensionElement object's properties define the location of the attributes that describe that element. A star model stores attributes in the same table as the columns corresponding to DimensionElement objects, whereas a snowflake model stores attributes in separate tables. MetaCube also supports partial snowflakes, in which the attributes describing some dimension elements are stored in separate tables, while the attributes describing other elements remain in the dimension table.

If attributes describing DimensionElement objects are stored in the same table as the element itself, you need not specify the DimensionElement properties that describe the location of these attributes. If the values of such properties as the **AttributeTable** property are null, MetaCube assumes the attributes are stored in the table identified by the Dimension object's **Table** property. Such properties, by default, are empty.

Figure 4-5 summarizes the properties of the DimensionElement object.

Figure 4-5
DimensionElements Class of Objects: Properties

Property	Description/Example
AggLevel	A string value, stored in the column identified by the Dimension object's **AggLevelColumn** property. Indicates at which rows tables can join to dimension tables to find distinct values for a given dimension element. This property can be specified or left empty (`NULL`): ■ When `NULL`, the **DimensionElementDET** values must be set. ■ When specified, the **Dimension.AggLevelColumn** must also be specified. `MyDimEl.AggLevel = "3"`
AttributeSchema	String. In snowflake or partial snowflake data models, the attributes describing a dimension element may be stored in tables separate from the dimension table. This attribute specifies the physical location / schema storing the attribute table for this dimension. `MyDimEl.AttributeSchema = "METADEMO"`
AttributeTable	String. In a snowflake model, the name of the attribute table for this dimension element. `MyDimEl.AttributeTable = "BRANDS"`

AttributeTo-DimensionColumn	String. In a snowflake model, the column in the attribute table used to join that table to the dimension table.
	`MyDimEl.AttributeToDimensionColumn = _`
	`"BRAND_CODE"`

Base	Boolean. True if the dimension element represents the lowest level in the dimensional hierarchy; false otherwise. `MyDimEl.Base = True`
DefaultAttribute	Object. Represents the default attribute for the dimension element; displayed when users drill up or down to that element. Read-only. `MsgBox MyDimEl.DefaultAttribute.Name`

(1 of 3)

Property	Description/Example
DETCreateOptions	String. CREATE TABLE statement options for the database where the data warehouse resides. `MyDimEl.DETCreate.Options="In MyDBSpace EXTENT SIZE 16"`
DETCreateStmt	String. SQL statements used to create and populate the dimension element table. `MyDimEl.DETCreateStmt=SQLString`
DETName	String. Name of the dimension element table. `MyDimEl.DETName="DET_Brand"`
DETSchemaName	String. Name of the dimension element table schema / owner. `MyDimEl.DETSchemaName="metacube"`
Dimension	Object. Identifies the Dimension object to which this element belongs. Read-only. `MsgBox MyDimEl.Dimension.Name`

DimensionTo-AttributeColumn	String. The name of the column in the dimension table storing the actual values of the dimension element. Also, in a snowflake model, the name of the column in the dimension table that joins that table to the attribute table.
	`MyDimEl.DimensionToAttributeColumn = _`
	`"BRAND_CODE"`

Name	String. Stores the name of the dimension element. Default property. `MsgBox MyDimEl.Name`

(2 of 3)

Property	Description/Example
Parent	Object. Identifies the Dimension object to which this element belongs. `MsgBox MyDimEl.Parent.Name`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a DimensionElement object is one of the following: ■ Completely valid ■ Invalid because at least one attribute describing the dimension element is invalid ■ Invalid because the DimensionElement object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. Verifying dimension elements that have not been incorporated into the dimensional hierarchy returns an error. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyDimEl.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the DimensionElement object's metadata. This will not include errors in the metadata for Attribute objects belonging to the DimensionElement object. `MsgBox MyDimEl.VerifyResults.TabbedValues`

(3 of 3)

DimensionElements Methods

The **DimensionElements** class of objects features only two methods, which either add or remove other elements from a dimension element's consolidation path in the dimensional hierarchy. The **AddDrillUp** method includes a dimension element in the DrillUps collection described on the following page, indicating that the level described by one dimension element exists directly below the hierarchy level of another dimension element.

For example, a DimensionElement object representing days may consolidate to DimensionElement objects representing weeks and months. In this example, weeks themselves do not consolidate to months, since there is not an even number of weeks in each month. Consequently, both weeks and month hierarchy levels sit directly above the day hierarchy level.

Deploying the **AddDrillUp** method requires you to specify as an argument the DimensionElement object to which you drill up, as shown in this section of sample code:

```
Set DimElLower = MyDimension.DimensionElements.Item("Day")
Set DimElHigher = MyDimension.DimensionElements.Item("Week")
DimElLower.AddDrillUp DimElHigher
```

To remove a DimensionElement object from another DimensionElement object's consolidation path, deploy the **RemoveDrillUp** method:

```
DimElLower.RemoveDrillUp DimElHigher
```

An **AddDrillDown** method does not exist, since MetaCube requires you to describe a hierarchy from the bottom up, such that a single Dimension-Element object sits at the base of any dimensional hierarchy.

DimensionElements Collections

Each level in a dimensional hierarchy may be described by many different attributes and may consolidate to several different hierarchy levels. For example, the Brand dimension element may be described by **Brand Name** and **Brand Manager** attributes. This element may consolidate to two different levels in the hierarchy, represented by Company and Product class elements.

The DimensionElement object collections represent the attributes describing a dimension element, the elements directly below the dimension element in the dimensional hierarchy, and the elements directly above the dimension element in the dimensional hierarchy. A DimensionElement object may thus store other DimensionElement objects in a collection. Dimension elements in a simple hierarchy will likely only contain one dimension element in these collections.

Figure 4-6
DimensionElement Class of Objects: Collections

Collection	Description/Example
Attributes	A collection of attributes describing the dimension element. `MsgBox MyDimEl.Attributes.Names`
DrillDowns	A collection of DimensionElement objects that exist at hierarchy levels directly below the parent. `MsgBox MyDimEl.DrillDowns.Names`
DrillUps	A collection of DimensionElement objects that exist at hierarchy levels directly above the parent. `MsgBox MyDimEl.DrillUps.Names`

The Attributes Class of Objects

The **Attributes** class of objects describes in MetaCube's metadata the descriptive categories by which users define queries. Attributes describe dimension elements and are hierarchically organized in the order of the dimension elements they describe.

The distinction between an Attribute object and a QueryCategory object, which the exercises prominently featured, could be confusing. To incorporate an attribute in the definition of a particular query, you must instantiate a QueryCategory object in a collection owned by a Query object. The **Attributes** class of objects differs from this QueryCategory class of objects insofar as its properties describe the physical structure of the database. A QueryCategory object can simply identify an Attribute object's name to enable MetaCube to generate SQL on the basis of the Attribute object's properties. The **Attributes** class of objects thus represents the library of attributes available for a query, whereas the QueryCategory object represents an attribute selected from that library for inclusion in a query's definition.

Although attributes describe a dimension element, you can view the attributes for a single dimension element or for all of the dimension elements in a dimension. Both Dimension and DimensionElement objects own collections of Attribute objects: the DimensionElement object's collection includes only those attributes that describe that dimension element; the Dimension object's collection of attributes owns that dimension element's attributes in addition to attributes describing other elements within the dimension. A DimensionElement object's collection of Attribute objects is thus a subset of the Dimension object's collection of Attribute objects.

The Attributes Collection's Add Method

Although an Attribute object belongs to both a collection owned by a Dimension object and a collection owned by a DimensionElement object, you must instantiate the object as a member of a dimension element's collection, the parent that the attribute actually describes. MetaCube requires two arguments in the instantiation command: the name of the attribute and the database column storing that attribute's values. This example instantiates an attribute named **Brand Name**, the values of which are stored in the column **BRAND_NAME**:

```
MyDimension.Attributes.Add "Brand", "BRAND_NAME"
```

The name of the database table to which this column belongs is specified as a property of the DimensionElement object, if your data model is a snowflake, or as a property of the Dimension object, if your data model is a star.

Attributes Properties

The properties of an Attribute object correspond to many of the fields in Warehouse Manager for an attribute, specifying the physical characteristics of the attribute, balloon help, sample values, and other information about the attribute. Aside from the standard **Verify** method, the Attribute object has no methods. Figure 4-7 summarizes the properties of the **Attributes** class of objects.

Figure 4-7
Attributes Class of Objects: Properties

Property	Description/Example
BalloonHelp	String. Stores a brief explanation of the attribute for popup balloon help in MetaCube Explorer and other applications. `MyAttribute.BalloonHelp = "Retail Stores"`
Column	String. The column in the attribute or dimension table storing the actual attribute values. `MyAttribute.Column = "BRAND_NAME"`
ColumnType	Integer. Identifies the type of data stored in the attribute column: character, numeric, date, or other. See constants below. `MyAttribute.ColumnType = DataTypeNumeric`
Default	Boolean. Indicates whether the attribute represents the default description of a dimension element. MetaCube Explorer displays the default attribute of a dimension element whenever a user drills up or down to that element, but other applications may use this information differently. Complements dimension element's read-only **DefaultAttribute** property. Defaults to false. Because there can only be one default attribute for any given dimension element, setting the **Default** property of one attribute to true reverts any other attributes within that dimension element's collection to false. `MyAttribute.Default = True`
Description	String. Stores a long description of the attribute for administrative purposes. `MyAttribute.Description = "Data from a legacy system..."`
Dimension	Object. The Dimension object to which the attribute belongs. Read-only. `MsgBox MyAttribute.Dimension.Name`
Dimension-Element	Object. The DimensionElement object described by the attribute; also the direct owner of the attribute's collection, as indicated by the **Parent** property. Read-only. `MsgBox MyAttribute.DimensionElement.Name`

(1 of 4)

Property	Description/Example
DisplayStyle	Long. Indicates whether the attribute is valid for display in query or filter interfaces. See the display constants listed in Figure 4-2 on page 4-8. `MyAttribute.DisplayStyle = DisplayStyleBoth`
Example1	String. Stores a sample value of the attribute in the metadata, which copies to client. MetaCube Explorer retrieves this value to populate a sample report before executing query. Defaults to empty. `MyAttribute.Example1 = "Alden"`
Example2	String. Stores a second sample value of the attribute. Defaults to empty
Example3	String. Stores a third sample value of the attribute. Defaults to empty.

ListSQL	String. Stores a custom SQL command retrieving the values of the attribute from the database, typically for display as a set of choices on which the user can filter.
	`MyAttribute.ListSQL = _`
	`"SELECT * FROM GENDER_VALUES"`

Name	String. Attribute name. Default property. `MyAttribute.Name = "Brand Name"`
Parent	Object. The DimensionElement object that owns the collection to which the attribute belongs. `MsgBox MyAttribute.Parent.Name`
ScreenOrder	Integer. Determines the order in which attributes appear in an interface; defaults to the order of instantiation. `MyAttribute.ScreenOrder = 1`

(2 of 4)

Property	Description/Example
SortColumn	String. Identifies the name of a numeric column other than the attribute column on which to perform a sort. The column specified is often a sequential dimension element. Instead of sorting attribute values directly, MetaCube can sort a set of values associated with each attribute. If, for example, the column representing a dimension element is identified as the sort column for an attribute, MetaCube sorts the attribute values based on the numeric values of the dimension element. Attribute values associated with a dimension element of a low numeric value appear first, and those associated with a dimension element of a high numeric value appear last. The attribute values continue to be displayed in a sorted report, but the column on which the sort is based do not. The **SortColumn** property is particularly useful for attributes of a time dimension, because different date formats can preclude MetaCube from directly sorting such attributes correctly. MetaCube can accurately sort dates regardless of date format by basing the sort on the sequential dimension element that enumerates each date. The column identified by this property must be in the same table as the column on which the attribute itself is based. By default, MetaCube sorts on the attribute value itself, in which case this property should remain empty or blank. `MyAttribute.SortColumn = "DAY_CODE"`
ValueList	ValueList. A list of values for this attribute, retrieved by MetaCube and stored in the metadata directly; allows applications to display filter choices rapidly. Read-only. `MsgBox MyAttribute.ValueList.TabbedValues`

(3 of 4)

Property	Description/Example
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for an Attribute object is either: ■ Completely valid ■ Invalid because the Attribute object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyAttribute.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the object's metadata. `MsgBox MyAttribute.VerifyResults.TabbedValues`

(4 of 4)

Figure 4-8 summarizes the constants stored by the **ColumnType** property of the **Attributes** class of objects.

Figure 4-8
ColumnType Properties

Data Type	MetaCons.bas Constant Name	Constant
Character	DataTypeChar	0
Numeric	DataTypeNumeric	1
Date	DataTypeDate	2
Other	DataTypeUnsupported	3

Attributes Collections

The collections of an Attribute object contain the attributes to which a user can drill from a value of the original attribute. The DrillUp collection consists of Attribute objects that you can directly reach by drilling up, whereas the DrillDown collection consists of Attribute objects that you can directly reach by drilling down.

The attributes included in each collection actually depend on dimension elements, which define the actual dimensional hierarchy. Associated with each dimension element is a default attribute, identified as such by the **Default** property of the Attribute object. When drilling to a new hierarchy level, a user cannot see all of the attributes describing a dimension element, only the default attribute for that dimension element. An attribute's drill collections thus contain only default attributes for dimension elements that sit either directly below or above the original attribute's dimension element in the hierarchy.

For example, a **Regional Manager** attribute might describe the Region level of detail in a dimensional hierarchy. Directly below the Region dimension element might sit District and City levels of detail, such that both cities and districts consolidate to regions but cities do not roll up to districts, or vice versa. If the default attributes for district and city are **City Name** and **District Name**, the Attribute objects representing **City Name** and **District Name** belong to a DrillDowns collection owned by the original Attribute object, named **Regional Manager**.

Figure 4-9 summarizes the Attribute object's collections.

Figure 4-9
Attributes Class of Objects: Collections

Collection	Description/Example
DrillDowns	A collection of **Attribute** objects that you can drill down to from the attribute. The attributes in this collection are limited to the default attributes for a set of dimension elements. In the dimensional hierarchy, this set of dimension elements sits directly above the dimension element described by the attribute. A simple hierarchy features only one element at the higher hierarchy level and one default attribute describing that element. `MsgBox MyAttribute.DrillDowns.Names`
DrillUps	A collection of **Attribute** objects that you can drill up to from the attribute. The attributes in this collection are limited to the default attributes for a set of dimension elements. In the dimensional hierarchy, this set of dimension elements sits directly below the dimension element described by the attribute. A simple hierarchy features only one element at the lower hierarchy level and one default attribute. `MsgBox.MyAttribute.DrillUps.Names`

Chapter

5

Extensions

In This Chapter

This chapter introduces the programming interface for incorporating extensions compiled in C++ into MetaCube. Once an extension has been registered through the programming interface, the functions within that extension can be deployed directly, as if the extension functions were native to MetaCube's analysis engine. This chapter also explains those functions that have been created as standard extensions available with any MetaCube release.

The Extensions Class of Objects

Once you have developed and compiled an extension, register the functions of that extension by instantiating an object of the **Extensions** class of objects. Registering an extension incorporates that extension's functionality into MetaCube until that extension is explicitly removed.

Although your instance of the **Extensions** object class might release when your application terminates, instantiating that object appends a permanent entry to MetaCube's initialization file, **metacube.ini**. This entry, called Enabled Extensions, identifies the filename and path of the extension and appears under the `[Engine]` header.

Whenever any application calls the MetaCube analysis engine, the engine automatically enables the extensions identified in the **metacube.ini** file. For each new extension, it is thus necessary to instantiate an object of the **Extensions** class only once.

The Extensions Collection's Add Method

To instantiate an object of the **Extensions** class, identify the parent Metabase object and that object's collection of Extension objects. Then deploy the **Add** method general to all collections. The **Add** method requires one argument, the filename of the extension:

```
MyMetabase.Extensions.Add "c:\metacube\mcplgmn.mcx"
```

The specified extension is registered in MetaCube's initialization file, and is automatically enabled upon all subsequent connections between an application and the MetaCube analysis engine. To disable an extension, deploy the **Remove** method, also general to all collections:

```
MyMetabase.Extensions.Remove 0
```

You can identify an extension by file name or by index number:

```
Set MyExtension = MyMetabase.Extensions.Item _
                    ("c:\metacube\mcplgmn.mcx")
```

Since referring to an Extension object by name is often inconvenient, you can store an instance of the object in an object variable such as MyExtension.

Extensions Properties

The properties of an object of the **Extensions** class store information about an extension for a programmer's referral, but few, if any, of these properties can be usefully incorporated into an application.

Figure 5-1
The Extensions Class of Objects: Properties

Property	Description/Example
Arguments	Stores a read-only ValueList of arguments required by each function, in the order in which they are to be specified. When retrieving arguments for a function, the name of the function for which arguments are to be retrieved must be specified. `MsgBox MyExtension.Arguments "FracOTot"`
Argument-Types	Stores a read-only ValueList indicating the type of each argument for a function, compiled in the same order as the arguments to which that list corresponds. When retrieving argument types for a function, the name of the function for which argument types are to be retrieved must be specified. `MsgBox MyExtension.ArgumentTypes "FracOTot"`
Description	Stores a read-only string description of the extension. `MsgBox MyExtension.Description`
Enabled	Stores a Boolean value indicating whether the extension has been enabled. `MsgBox MyExtension.Enabled`
FileName	Stores the file name of the compiled extension code as a read-only string. `MsgBox MyExtension.FileName`
Functions	Stores a read-only ValueList of the functions included in an extension. `MsgBox MyExtension.Functions`
Types	Stores a read-only ValueList of the object classes to which the function applies, typically either the string “QueryCategory” or “QueryItem.” `MsgBox MyExtension.Types`

Because many of the properties of the **Extensions** object class return ValueLists including the names or arguments of all functions, application developers might not want to compare such lists against one another to discover the arguments of a particular function.

Exercise 14 on page 5-7 generates a report listing the name of each function, the object class to which that function applies, the arguments required by that function, and the argument types. This program can evaluate any extension, provided the name of the extension and the number of functions included in that extension are accurately specified by the constants **ExtensionName** and **NumberofFunctions**.

As shown, the program displays information about the functions of the main MetaCube extension. As subsequent exercises deploy these functions, the report generated by this application can be a useful reference when you read later sections of this manual.

Exercise 14: Displaying Functions Within an Extension and Displaying Arguments for Those Functions

```
1  Option Explicit
2
3  'MetaCube API Exercise 14: Displaying Functions within an
   Extension
4  '    and Displaying Arguments for Those Functions
5
6   Sub MetaCube_API()
7
8  'Declare Constants
9      Const ExtensionName = "c:\MetaCube\mcplgmn.mcx"
10     Const NumberofFunctions = 16
11
12 'Declare Variables
13     Dim MyMetabase As Object, _
14     MyExtension As Object, Count As Integer, _
15     FunctionNames As Variant, _
16     FunctionTypes As Variant, _
17     Arguments As String, _
18     ArgumentTypes As String
19
20 'Connect
21     Set MyMetabase = CreateObject("Metabase")
22
23     'Identify an ODBC Data Source
24     Let MyMetabase.ConnectString = "Metademo"
25
26     'Specify a set of metadata
27     Let MyMetabase.Name = "MetaCube Demo"
28
29     'Specify login to database
30     Let MyMetabase.Login = "metademo"
31
32     'Specify database password
33     Let MyMetabase.Password = "Metademo"
34
35     MyMetabase.Connect
36
37 'Enable this Extension, If Necessary
38 '     Set MyExtension = _
39 '         MyMetabase.Extensions.Add(ExtensionName)
40
41     Set MyExtension = MyMetabase.extensions.Item(0)
42
43 'Get Function Names and Types
44     Let FunctionNames = _
```

```
45          MyExtension.Functions.ArrayValues
46      Let FunctionTypes = _
47          MyExtension.Types.ArrayValues
48
49  'Report Headers
50      Worksheets.Item("Query Report").Activate
51      ActiveSheet.Cells(1, 1) = "Function Name"
52      ActiveSheet.Cells(1, 2) = "Function Type"
53      ActiveSheet.Cells(1, 3) = "Arguments"
54      ActiveSheet.Cells(1, 4) = "Argument Types"
55  'Cycle Through Functions
56  For Count = 0 To NumberofFunctions
57      Let Arguments = _
58          MyExtension.Arguments(FunctionNames(Count))
59      Let ArgumentTypes = _
60          MyExtension.ArgumentTypes(FunctionNames(Count))
61      ActiveSheet.Cells(Count + 2, 1) = _
62          FunctionNames(Count)
63      ActiveSheet.Cells(Count + 2, 2) = _
64          FunctionTypes(Count)
65      ActiveSheet.Cells(Count + 2, 3) = Arguments
66      ActiveSheet.Cells(Count + 2, 4) = ArgumentTypes
67  Next Count
68
69  'Format Report
70      ActiveSheet.Range(ActiveSheet.Cells(1, 1), _
71          ActiveSheet.Cells(Count + 1, 4)).Select
72      Selection.EntireColumn.AutoFit
73
74  End Sub
```

The Main MetaCube Extension Functions

Included with every MetaCube executable is the main MetaCube extension, a set of standard analytical functions called by MetaCube applications such as MetaCube Explorer and MetaCube for Excel. The file for this extension, **mcplgmn.mcx,** can be found in the MetaCube directory. Once this extension has been enabled, you can replace attribute and measure names with complex expressions when instantiating QueryItems and QueryCategories.

As discussed in "The QueryCategories Class of Objects" on page 8-22, QueryCategory and QueryItem objects typically refer to the names of attributes and measures, as defined in MetaCube's metadata by **Attribute** and **Measure** objects:

```
MyQuery.QueryItems.Add "Brand"
```

The functions included in an extension perform operations on attribute and measure values, manipulating or preprocessing data before it becomes incorporated as an attribute or a measure in MetaCube's virtual cube of data.

An extension's functions can only return values to a QueryCategory or a QueryItem object, and these functions can only be used as expressions when instantiating such objects. When instantiating a QueryItem, the syntax for such an expression is straightforward, with the name of the function and all arguments enclosed in quotation marks, in the same position as the name of a measure. Any arguments required by the function follow the function name, in a list enclosed in parentheses. The syntax is of the general type:

```
MyQuery.QueryItems.Add "FUNCTION_NAME _
    (Argument 1, Argument 2, Argument 3...)"
```

No general syntax has been formulated for QueryCategory expressions. For an explanation of the two attribute functions available in the main MetaCube extension, see "Extension Functions as QueryCategory Expressions" on page 5-42.

The next section explains the functions that can be deployed as QueryItem expressions.

Extension Functions as QueryItem Expressions

The functions included in MetaCube's main extension enable complex statistical comparisons, summations, and averages. Two functions, **TOPN** and **TOPPCT**, limit the number of rows displayed in a report to those rows associated with the highest or lowest values of a measure. The **TOPN** function replaces the **TopN** object class, which in previous releases belonged to the **MetaCube** object class.

To help you understand the operations a function performs on the values of a measure, subsequent sample procedures always include as a separate QueryItem the measure on which the function is being performed. Such pairing is, however, unnecessary. A measure that is otherwise excluded from the query can be included in an expression.

For example, a query can return both gross revenues by quarter and the percentage change in operating costs, although gross revenue is a simple measure, and percentage change is a function performed on a second measure, operating costs. In fact, a query could simply return percentage change in operating costs without including any other measures.

Figure 5-2 describes each of the functions in the main MetaCube extension that apply to measures. In each description measures are organized by columns, their default orientation. A longer explanation of each function follows, discussing how pivoting attributes and measures alters the result returned by the function.

Figure 5-2
Functions as QueryItem Expressions

Function	Description/Example
ABS_CHANGE	Compares the difference between columns of data. `MyQuery.QueryItems.Add "ABS_CHANGE (Units Sold)"`
FracGTot	Calculates the fraction of a value compared to the sum of all values for every page, column, and row in the report. This fraction can be multiplied by any factor, as specified in the second argument. `MyQuery.QueryItems.Add "FracGTot (Units Sold, 100)"`
FracOTot	Calculates the fraction of a value compared to the sum for the entire row. This fraction can be multiplied by any factor, as specified in the second argument. `MyQuery.QueryItems.Add "FracOTot (Units Sold, 100)"`
FracPTot	Calculates the fraction of a value compared to the sum for the entire page. This fraction can be multiplied by any factor, as specified in the second argument. `MyQuery.QueryItems.Add "FracPTot (Units Sold, 100)"`

FracSTot	Calculates the fraction of a value compared to the sum of the subtotal for that value. Arguments: the measure, the multiplier, and the attribute on which subtotals are being performed.
	`MyQuery.QueryItems.Add _`
	`"FracSTot (Units Sold, 100, Brand)"`

FracTot	Calculates the fraction of a value compared to the total for a column. This fraction can be multiplied by any factor, as specified in the second argument. `MyQuery.QueryItems.Add "FracTot (Units Sold, 100)"`
MovingAvg	Calculates the average of a value and a specified number of values in preceding rows of the same column. Arguments: measure name and number of values over which to average. `MyQuery.QueryItems.Add "MovingAvg(Units Sold, 2)"`
MovingSum	Calculates the sum of a value and a specified number of values in preceding rows of the same column Arguments: measure name and number of preceding values to sum. `MyQuery.QueryItems.Add "MovingSum (Units Sold, 2)"`

(1 of 2)

Function	Description/Example
PCT_CHANGE	Calculates the percent change between columns of data. `MyQuery.QueryItems.Add "PCT_CHANGE (Units Sold)"`
PCT_PREV	Calculates a value as a percentage of the value in the preceding column of data. `MyQuery.QueryItems.Add "PCT_PREV (Units Sold)"`
QUANTILE	For each column, divides values of a column into a specified number of ranked categories according to their magnitude. Arguments: measure name and number of quantiles or categories. `MyQuery.QueryItems.Add "QUANTILE (Units Sold, 3)"`
RUNNINGSUM	Calculates the sum of a value and all values in preceding rows of the same column. `MyQuery.QueryItems.Add "RUNNINGSUM (Units Sold)"`
TOPN	Limits the rows displayed to those associated with the highest or lowest values of a measure. The number of rows in the report is set as an absolute numeric argument. Arguments: measure name, number of rows to display, column number, and ascending / descending flag. `MyQuery.QueryItems.Add "TOPN (Units Sold, 3, 0, Asc)"`
TOPPCT	Limits the rows displayed to those associated with the highest or lowest values of a measure. This function only displays those rows storing values that are within a specified percentage of the highest value in a row or column. Arguments: measure name, percentage, column number, and ascending / descending flag. `MyQuery.QueryItems.Add "TOPPCT(Units Sold, 34, 0, Asc)"`

(2 of 2)

The Absolute Change Function

This function compares two or more columns or rows of the same measure, interpolating an additional measure indicating the difference between the two. The **ABS_CHANGE** function only compares columns or rows of data measured in the same units, ignoring intervening columns or rows measured in different units.

If a query's measures are organized by columns, this function compares numeric data as organized by columns and thus depends on the orientation of at least one attribute by columns. A minimum of two columns of data must be retrieved for the comparison to function. Conversely, if the orientation of measures is by rows, an attribute with at least two values must be also be organized by rows. The only argument for this function is the name of the measure on which it is being performed:

```
MyQuery.QueryItems.Add "ABS_CHANGE (MEASURE NAME)"
```

This function can be performed on measures otherwise excluded from the query and its resulting report. In such cases, MetaCube retrieves numeric values for the measure on which the function is being performed but only displays the result of the function. Exercise 15 develops a simple query using the main MetaCube extension's absolute change function.

Exercise 15: The Absolute Change Function

```
1  Option Explicit
2
3  'MetaCube API Exercise 15: The Absolute Change Function
4
5   Sub MetaCube_API()
6
7  'Declare Variables
8      Dim MyMetabase As Object, MyQuery As Object, _
9          MyMetaCube As Object, MyData As Variant
10
11 'Excel Variables
12     Dim ReportRange As Range, _
13          MyListBox As ListBox
14
15 'Declare Constants
16     Const FirstAttribute = "Brand"
17     Const FirstPivot = 1   'By Rows
18     Const SecondAttribute = "Region"
19     Const SecondPivot = 2 'By Columns
20     Const Expression = "ABS_CHANGE (Units Sold)"
21     Const MeasurePivot = 2 'By Columns
22
23 'Connect
24     Set MyMetabase = CreateObject("Metabase")
25
26     'Identify an ODBC Data Source
27     Let MyMetabase.ConnectString = "Metademo"
28
29     'Specify a set of metadata
30     Let MyMetabase.Name = "MetaCube Demo"
31
32     'Specify login to database
33     Let MyMetabase.Login = "metademo"
34     'Passes value to database on connection
35
36     'Specify database password
37     Let MyMetabase.Password = "Metademo"
38
39     'Enable this Extension if necessary
40     'MyMetabase.extensions.Add _
41     '    "c:\metacube\mcplgmn.mcx"
42     MyMetabase.Connect
43
44 'Define Query
45     Set MyQuery = MyMetabase.Queries.Add _
46         ("My New Query")
47
```

```
48      'QueryCategories
49          MyQuery.QueryCategories.Add FirstAttribute
50          MyQuery.QueryCategories.Add SecondAttribute
51
52      'QueryItems
53          MyQuery.QueryItems.Add "Units Sold"
54          MyQuery.QueryItems.Add Expression
55
56  'Pivoting
57      MyQuery.QueryCategories.Item(0).Orientation = _
58          FirstPivot
59      MyQuery.QueryCategories.Item(1).Orientation = _
60          SecondPivot
61      MyQuery.ItemOrientation = MeasurePivot
62
63  'Get Results
64      Worksheets.Item("Query Report").Activate
65      Cells.Select
66      Selection.ClearContents
67      Set MyMetaCube = MyQuery.MetaCubes.Add("Data")
68      Let MyData = MyMetaCube.ToVBArray
69      Set ReportRange = ActiveSheet.Range _
70          (ActiveSheet.Cells(1, 1), _
71          ActiveSheet.Cells _
72          (MyMetaCube.Rows, MyMetaCube.Columns))
73      Let ReportRange.Value = MyData
74      ReportRange.EntireColumn.AutoFit  'Sizes columns
75
76  End Sub
```

Executing this procedure generates the report displayed in Figure 5-3.

Figure 5-3
Result of Exercise 15: Brand by Rows, Region by Columns, Measures by Columns

Region	Northeast	West	West
Brand	Units Sold	Units Sold	ABS_CHANGE (Units Sold)
Alden	1811	2626	815
Barton	1314	1924	610
Delmore	1778	2557	779

Region	Northeast	West	West
Extreme	433	649	216
Lasertech	1105	1665	560
NVD	2719	3788	1069
Onetron	910	1254	344
Suresound	2548	3464	916
Techno	3699	5286	1587

Fraction of Grand Total

For each numeric value of a specified measure within a report, the **FracGTot** function calculates the fraction of that value compared to the sum of that measure or expression for all cells, in columns, rows, and pages of the report, multiplying the fraction by a numeric factor specified as an argument to the function.

To see fractional values as a percentage, choose a factor of 100. You must also specify the name of the measure or expression on which the calculation is being performed:

```
MyQuery.QueryItems.Add"FracGTot(MEASURE NAME, MULTIPLIER)"
```

As with **ABS_CHANGE**, this function can be performed on measures not otherwise included in the query. In such cases, the data retrieved to perform the calculation is not displayed in the report.

By substituting different constant values in Exercise 15, you can develop an application incorporating this function:

```
Declare Constants

    Const FirstAttribute = "Brand"
    Const FirstPivot = 1    'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2  'By Columns

    Const Expression = "FracGTot (Units Sold, 100)"
    Const MeasurePivot = 2 'By Columns
```

Substituting this code in lines 15 through 21 of Exercise 15 and executing the altered application generates the report displayed in Figure 5-4. The values in the **FracGTot** cells in all rows of both columns sum to 1, multiplied by 100, the factor specified as the second argument in the **FracGTot** function.

Figure 5-4
Fraction of Grand Total for a Brand-Region Query

Region	Northeast	Northeast	West	West
Brand	Units Sold	FracGTot (Units Sold, 100)	Units Sold	FracGTot (Units Sold, 100)
Alden	1811	4.581330635	2626	6.643055907
Barton	1314	3.324057678	1924	4.867189476
Delmore	1778	4.497849734	2557	6.468504933
Extreme	433	1.095370605	649	1.641791045
Lasertech	1105	2.795345307	1665	4.211990893
NVD	2719	6.878320263	3788	9.582595497
Onetron	910	2.302049077	1254	3.172274222
Suresound	2548	6.445737415	3464	8.762964837
Techno	3699	9.357450038	5286	13.37212244

Fraction of Orthogonal Total

For each numeric value of a specified measure or expression within a report, the **FracOTot** function calculates the fraction of that value compared to all values of that measure or expression within the same row. When measures are organized by column, this function depends on the orientation of at least one attribute by columns; a minimum of two columns of data must be retrieved for the comparison to function. The **FracOTot** function requires two arguments, the name of the measure or the expression, and the factor by which to multiply the fraction. As before, you need not include in the query the measure on which the function is being performed.

```
MyQuery.QueryItems.Add "FracOTot(MEASURE NAME, MULTIPLIER)"
```

Pivoting measures to the *row* orientation directs the **FracOTot** function to calculate the fraction of each measure's value compared to all values of that measure within the same *column*. As this function's full name—fraction of orthogonal total—implies, this function always derives fractions from the sum of values appearing orthogonal to its own orientation as a measure.

By substituting different constant values in Exercise 15, you can develop an application incorporating the **FracOTot** function:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1    'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2  'By Columns

    Const Expression = "FracOTot (Units Sold, 100)"
    Const MeasurePivot = 2 'By Columns
```

The report generated by the altered application is displayed in Figure 5-5. In this report, the sum of the **FracOTot** values in each row total 1, multiplied by 100, the factor specified as the second argument in the function. The fraction calculated is thus the fraction of the total for each row, even though measures have been organized by column.

Figure 5-5
Fraction of Orthogonal Total for a Brand-Region Query

Region	Northeast	Northeast	West	West
Brand	Units Sold	FracOTot (Units Sold, 100)	Units Sold	FracOTot (Units Sold, 100)
Alden	1811	40.81586658	2626	59.18413342
Barton	1314	40.58060531	1924	59.41939469
Delmore	1778	41.01499423	2557	58.98500577
Extreme	433	40.01848429	649	59.98151571
Lasertech	1105	39.89169675	1665	60.10830325
Suresound	2548	42.38190286	3464	57.61809714
Techno	3699	41.16861436	5286	58.83138564

Fraction of Page Total

The **FracPTot** function performs the same calculation as the **FracGTot** function but compares a single value of a measure or an expression to the total of that measure or expression for an entire page, rather than the entire query result. If you have not subdivided a query result into different pages, the **FracPTot** function returns the same values as the **FracGTot** function. As with previous functions, you must specify the measure or expression for which you are calculating the fraction and the factor by which that fraction should be multiplied.

```
MyQuery.QueryItems.Add "FracPTot(MEASURE, MULTIPLIER)"
```

Since Excel cannot display more than a page of results, this function cannot be meaningfully incorporated into a Visual Basic for Applications procedure.

Fraction of Subtotal

The **FracSTot** function calculates each value of a specified measure or expression as a fraction of a subtotal for that measure or expression. When measures are organized by columns, a subtotal sums numeric values for a group of rows.

For example, if you subdivide an attribute such as **Brand** by region, you can then sum each brand's total sales for all regions. Instantiating a Summary object, as explained in "The Summaries Class of Objects" on page 8-76, interpolates subtotals for each brand.

Within a subtotal, the **FracSTot** function calculates each value as a fraction of that subtotal; if **Brand** sales are subdivided by region, this function calculates regional brand sales as a fraction of the total sales for that brand. You need not include raw sales data or the subtotals of sales to calculate the fractional subtotal for sales.

The function does, however, require that the query organize multiple attributes by rows so that there are clear subdivisions by which to calculate the subtotal and fractional subtotal. The **FracSTot** function requires three arguments: the name of the measure or expression on which to base the calculation, the factor by which to multiply the fraction, and the name of the attribute that constitutes the broader grouping in the report.

```
MyQuery.QueryItems.Add "FracSTot(MEASURE NAME, MULTIPLIER,
NAME OF THE SUBDIVIDED ATTRIBUTE)"
```

Pivoting measures to the row orientation directs this function to calculate subtotals on the basis of attributes organized as columns and subcolumns.

With measures in their standard columnar orientation, the **FracSTot** function bases its calculation on fractions of the sums of values within a row, at break points defined by the attribute instantiated first. The constant values substituted in Exercise 15 thus organize both attributes by row, leaving measures in the column orientation:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1    'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 1  'By Rows, i.e. subrows

    Const Expression = _
                "FracSTot (Units Sold, 100, Brand)"
    Const MeasurePivot = 2  'By Columns
```

Executing the altered procedure generates a report similar to the one shown in Figure 5-6. To illustrate the premise of the **FracSTot** function, the code has further been altered to include the actual subtotal, although such a modification is unnecessary for the function to calculate fractional values and need not be included in your own programs. To learn how to include subtotals in a report, see "The Summaries Class of Objects" on page 8-76.

Figure 5-6
Fraction of Subtotal for a Brand-Region Query

Brand	Region	Units Sold	FracSTot (Units Sold, 100, Brand)
Alden	Northeast	1811	40.81586658
Alden	West	2626	59.18413342
Alden	Total	4437	100
Barton	Northeast	1314	40.58060531
Barton	West	1924	59.41939469
Barton	Total	3238	100
Delmore	Northeast	1778	41.01499423
Delmore	West	2557	58.98500577

(1 of 2)

Brand	Region	Units Sold	FracSTot (Units Sold, 100, Brand)
Delmore	Total	4335	100
Extreme	Northeast	433	40.01848429
Extreme	West	649	59.98151571
Extreme	Total	1082	100
Suresound	Northeast	2548	42.38190286
Suresound	West	3464	57.61809714
Suresound	Total	6012	100
Techno	Northeast	3699	41.16861436
Techno	West	5286	58.83138564
Techno	Total	8985	100

(2 of 2)

In Figure 5-6, the two regional values of Northeast and West subdivide eight brands, with the **FracSTot** function calculating the fraction each region contributes to the brand's total sales for both regions. For the subtotals interpolated at each break point, the **FracSTot** function always returns a value of 100.

Fraction of Total

The **FracTot** function calculates the value of a measure or expression as a fraction of the total values in a column or row, multiplying that fraction by a factor such as 100 to return a percentage. With measures in their default orientation as columns, this function operates on measures as they would appear in a column, comparing one value to its total for the column. When measures have been pivoted to rows, the function compares a value to its total for the row in which it would appear. As always, the measure on which the function is being performed need not be included in the query definition. The function requires the same arguments as the **FracOTot** and **FracGTot** functions, explained above:

```
MyQuery.QueryItems.Add "FracTot(MEASURE, MULTIPLIER)"
```

This function and the **FracOTot** function can be thought of as operating at right angles to one another, such that pivoting measures cause the **FracTot** function to return the same values as calculated by the **FracOTot** function for the original query.

To illustrate this point, substitute the same measures, attributes, and attribute orientations in procedures featuring **FracOTot** and **FracTot** expressions, with the only difference being in the orientation of measures:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1    'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2  'By Columns

    Const Expression = "FracTot (Units Sold, 100)"
    Const MeasurePivot = 1 'By Rows
```

Substituting the lines above for similar lines in Exercise 15 generates the report displayed in Figure 5-7 on page 5-24. Comparing this result to Figure 5-5 on page 5-19 confirms that the two functions return the same values when pivoted to opposite orientations.

Figure 5-7
Fraction of Total For a Brand-Region Query, With Measures Pivoted to the Row Orientation

Brand	Region	Northeast	West
Alden	Units Sold	1811	2626
Alden	FracTot (Units Sold, 100)	40.81586658	59.18413342
Barton	Units Sold	1314	1924
Barton	FracTot (Units Sold, 100)	40.58060531	59.41939469
Delmore	Units Sold	1778	2557
Delmore	FracTot (Units Sold, 100)	41.01499423	58.98500577
Extreme	Units Sold	433	649
Extreme	FracTot (Units Sold, 100)	40.01848429	59.98151571
Lasertech	Units Sold	1105	1665
Lasertech	FracTot (Units Sold, 100)	39.89169675	60.10830325
NVD	Units Sold	2719	3788
NVD	FracTot (Units Sold, 100)	41.78576917	58.21423083
Onetron	Units Sold	910	1254
Onetron	FracTot (Units Sold, 100)	42.05175601	57.94824399
Suresound	Units Sold	2548	3464

(1 of 2)

Brand	Region	Northeast	West
Suresound	FracTot (Units Sold, 100)	42.38190286	57.61809714
Techno	Units Sold	3699	5286
Techno	FracTot (Units Sold, 100)	41.16861436	58.83138564

(2 of 2)

Moving Average

The **MovingAvg** function averages a measure or expression with preceding values of that measure or expression. If measures are organized by columns, the preceding values of that measure or expression are taken from higher cells of the same column; if measures are organized by rows, the preceding values are taken from cells to the left in the same row. For the initial cell on which an average is to be calculated, there are no preceding cells, so the moving average is simply the value of that cell for the specified measure. Along with the name of the measure or expression being averaged, the number of preceding values over which to calculate the average is specified as an argument to the function. As always, the measure on which the function is being performed need not itself be included in the query.

```
MyQuery.QueryItems.Add "MovingAvg (MEASURE, # OF VALUES)
```

To demonstrate this function, we can substitute the following constant declarations into Exercise 15, with the **MovingAvg** function as the expression:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Fiscal Week"
    Const SecondPivot = 2    'By Columns

    Const Expression = "MovingAvg (Units Sold, 3)"
    Const MeasurePivot = 1   'By Row
```

Because moving averages are often calculated as a measure changes over time, the **Fiscal Week** attribute replaces the **Brand** attribute in this query and in the query for the **MovingSum** function. Figure 5-8 displays the data returned by the modified procedure, including only 4 of the 26 weeks displayed in the complete report.

Figure 5-8
Moving Average Function for a Brand-Fiscal Week Query (Four Weeks Only)

Brand	Fiscal Week	94/01/01 - 94/01/07	94/01/08 - 94/01/14	94/01/15 - 94/01/21	94/01/22 - 94/01/28
Alden	Units Sold	161	141	135	147
Alden	MovingAvg (Units Sold, 3)	161	151	145.6666	141
Barton	Units Sold	115	118	96	96
Barton	MovingAvg (Units Sold, 3)	115	116.5	109.6666	103.3333
Delmore	Units Sold	186	162	161	135
Delmore	MovingAvg (Units Sold, 3)	186	174	169.6666	152.6666
Extreme	Units Sold	*46*	***34***	***27***	**40**
Extreme	MovingAvg (Units Sold, 3)	46	40	*35.66666*	**33.66666**
Lasertech	Units Sold	108	93	83	93
Lasertech	MovingAvg (Units Sold, 3)	108	100.5	94.66666	89.66666
Techno	Units Sold	360	320	294	268
Techno	MovingAvg (Units Sold, 3)	360	340	324.6666	294

Since this query organizes measures by rows, the **MovingAvg** function incorporates preceding values from the same row to calculate the average. For example, to calculate the average brand sales of Extreme over three weeks for the week of January 15-21, the function averages sales for the week of January 1, the week of January 8, and, of course, the week of January 15. These values appear in italics.

Similarly, to calculate the average weekly sales over a three-week period ending the week of January 22-28, the function averages sales for the week of January 8, the week of January 15, and the week of January 22. These values appear in boldface. As the function iterates through the weeks, the same value may be included in several averages, which explains why sales for several weeks are displayed in both italics and boldface.

For the first two weeks of data, the function cannot evaluate the average over three weeks, so it calculates the average for as many weeks as are available.

Moving Sum

The **MovingSum** function sums a measure or expression with the preceding values of that measure or expression. The number of preceding values included in the sum is specified as an argument to the function. For each cell in a report for which the moving sum is to be calculated, the preceding values are taken to be the values for the measure or expression that would appear in higher cells of the same column. If measures have been pivoted to a row orientation, the preceding values are taken to be the values for the measure or expression that would appear in the same row and to the left. As always, the measure on which the function is being performed need not be included in the query.

```
MyQuery.QueryItems.Add "MovingSum (MEASURE, # OF VALUES)"
```

Substituting constant expressions in Exercise 15 generates the report displayed in Figure 5-9 on page 5-28:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Fiscal Week"
    Const SecondPivot = 2    'By Columns

    Const Expression = "MovingSum (Units Sold, 3)"
    Const MeasurePivot = 1   'By Rows
```

As before, the **Fiscal Week** attribute has been substituted for **Region**, since moving sums are often calculated over time. Since time periods often appear as different columns in a report, attributes and measures have been pivoted such that the moving sum function sums values appearing in the same row, but preceding columns. Only 2 of the 26 weeks appearing in the complete report are included in Figure 5-9.

Figure 5-9
Moving Sum Function in a Brand-Fiscal Week Query (Four Weeks Only)

Brand	Fiscal Week	94/01/01 - 94/01/07	94/01/08 - 94/01/14	94/01/15 - 94/01/21	94/01/22 - 94/01/28
Alden	Units Sold	161	141	135	147
Alden	MovingSum (Units Sold, 3)	161	302	437	423
Barton	Units Sold	115	118	96	96
Barton	MovingSum (Units Sold, 3)	115	233	329	310
Delmore	Units Sold	186	162	161	135
Delmore	MovingSum (Units Sold, 3)	186	348	509	458
Extreme	Units Sold	*46*	***34***	***27***	**40**
Extreme	MovingSum (Units Sold, 3)	46	80	*107*	**101**
Lasertech	Units Sold	108	93	83	93
Lasertech	MovingSum (Units Sold, 3)	108	201	284	269
NVD	Units Sold	259	239	214	190
NVD	MovingSum (Units Sold, 3)	259	498	712	643
Techno	Units Sold	360	320	294	268
Techno	MovingSum (Units Sold, 3)	360	680	974	882

In this example, the **MovingSum** function calculates at weekly intervals the sum of the past three weeks' sales. If, instead of the **Fiscal Week** attribute, the **Region** attribute had been organized by columns, the function would have returned for each region the sum of that and the two preceding regions' sales. The order in which regions are sorted determines the values returned by the function.

In cells for which the specified number of preceding values is not available, the **MovingSum** function returns a partial sum. Indeed, for the first cell in each row, the function sums one value, returning unchanged the value of the measure on which the function operates.

For the Extreme brand in the third week of the report, January 15-21, the function sums sales for that week and the two preceding weeks. In the fourth week, the function reaches the maximum number of values allowed by the second argument of the expression, 3, and excludes the first week's sales from the sum to include the fourth week's sales. Values that are included in the third week's moving sum are italicized, and values included in the fourth week's moving sum appear in boldface.

Percent Change

The **PCT_CHANGE** function calculates the percent change between two values of a measure or an expression, comparing every two values that appear in each row of a report according to the formula:

```
((Second Value/First Value) - 1) *100
```

Taking the first of two measure values as 100 percent, the function evaluates whether the second measure value is greater or smaller than the first and by what percentage.

Because the function compares columns of data, it returns no values for the first column of data since the function cannot compare that column to any preceding columns.

If measures have been pivoted to the row orientation, this function calculates the percentage change between each pair of cells within the same column of a report, returning its first result for the second rather than the first row of data.

The **PCT_CHANGE** function requires no arguments except the name of the measure on which the function operates.

```
MyQuery.QueryItems.Add "PCT_CHANGE (MEASURE)"
```

Substituting new constant values at the indicated line numbers in Exercise 15 enables you to quickly develop an application demonstrating this function:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2    'By Columns

    Const Expression = "PCT_CHANGE (Units Sold)"
    Const MeasurePivot = 2   'By Columns
```

Executing the modified procedure returns the report displayed in Figure 5-10.

Figure 5-10
Percent Change Function for a Brand-Region Query

Region	Northeast	West	West
Region	Northeast	West	West
Brand	Units Sold	Units Sold	Pct_Change (Units Sold)
Alden	1811	2626	45.00276091
Barton	1314	1924	46.42313546
Delmore	1778	2557	43.81327334
Extreme	433	649	49.88452656
Lasertech	1105	1665	50.67873303
NVD	2719	3788	39.31592497
Onetron	910	1254	37.8021978
Suresound	2548	3464	35.94976452
Techno	3699	5286	42.90348743

That all the values returned by the function are positive indicates that sales for the West region are uniformly better than for the Northeast region. For the Extreme brand, for example, the values returned by the **PCT_CHANGE** function indicate that the West's sales for that brand are nearly 50 percent greater than the Northeast's sales for that brand.

Reversing the sort of the report so that West appeared first and Northeast second would change the values returned by the function. In fact, all values returned by the function would be negative, since the column associated with larger values would appear first.

Percent of Previous

The **PCT_PREV** function calculates a value of a measure or an expression as a percentage of the previous value in the same row according to the formula:

```
(Second Value/First Value) *100
```

The formula of this function differs only slightly from the formula of the **PCT_CHANGE** function. Both functions calculate the fraction of one value as compared to a previous value, but the **PCT_CHANGE** formula subtracts 1 from the fraction before converting to a percentage. When comparing two values, the **PCT_CHANGE** function returns a change, for example, of -20 percent, whereas the **PCT_PREV** function in this example returns a percentage, 80, indicating that the second value is 80 percent of the first.

Like the **PCT_CHANGE** function, the **PCT_PREV** function compares by default each column of data to its predecessor, returning no values for the first column of data. Pivoting measures prompts this function to compare values across rows rather than columns. The **PCT_PREV** function requires one argument, the name of the measure or expression on which the function is being performed:

```
MyQuery.QueryItems.Add "PCT_PREV (MEASURE)"
```

Substituting new constant expressions in Exercise 15 enables you to incorporate this new function in a procedure:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2    'By Columns

    Const Expression = "PCT_PREV(Units Sold)"
    Const MeasurePivot = 2   'By Columns
```

Executing the procedure generates the report displayed in Figure 5-11. Comparing the values returned by the **PCT_CHANGE** and **PCT_PREV** functions in Figure 5-10 and Figure 5-11 confirms that the difference between the results of the two functions is always 100.

Figure 5-11
Percent of Previous for a Brand-Region Query

Region	Northeast	West	West
Brand	Units Sold	Units Sold	PCT_PREV (Units Sold)
Alden	1811	2626	145.0027609
Barton	1314	1924	146.4231355
Delmore	1778	2557	143.8132733
Extreme	433	649	149.8845266
Lasertech	1105	1665	150.678733
NVD	2719	3788	139.315925
Onetron	910	1254	137.8021978
Suresound	2548	3464	135.9497645
Techno	3699	5286	142.9034874

Quantiles

Where the value of a measure or an expression would normally appear in a report, the **QUANTILE** function assigns a categorical ranking on the basis of that value, indicating the quantile to which that value belongs compared to other values in the same column. If, for example, you were to rank the sales of nine brands into three quantiles (tertiles, to be precise), the brands would be divided into three categories on the basis of their sales, with the top three selling brands assigned a value of 1, the next three a value of 2, and the last three a value of 3.

The function requires you to specify as arguments the measure or expression on which the calculation is based, and the number of quantiles by which to categorize the values of that measure.

```
MyQuery.QueryItems.Add "QUANTILE (MEASURE, # OF CATEGORIES)"
```

Pivoting measures to a row orientation directs this function to compare the values of a measure or an expression to other values appearing in the same row rather than in the same column.

Substituting new constant expressions in Exercise 15 enables you to incorporate this function as an expression in the standard query on sales, by the **Brand** attribute and by the **Region** attribute.

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2    'By Columns

    Const Expression = "QUANTILE (Units Sold, 3)"
    Const MeasurePivot = 2   'By Columns
```

Executing the modified procedure creates the report displayed in Figure 5-12 on page 5-34. In that report, sales of nine brands are evenly divided into three ranked groups. If row values cannot be evenly divided between the specified number of categorical rankings, the function assigns lower rather than higher rankings, as necessary.

Figure 5-12
Quantile Function for a Brand-Region Query

Region	Northeast	Northeast	West	West
Brand	Units Sold	QUANTILE (Units Sold, 3)	Units Sold	QUANTILE (Units Sold, 3)
Alden	1811	2	2626	2
Barton	1314	2	1924	2
Delmore	1778	2	2557	2
Extreme	433	3	649	3
Lasertech	1105	3	1665	3
NVD	2719	1	3788	1
Onetron	910	3	1254	3
Suresound	2548	1	3464	1
Techno	3699	1	5286	1

Running Sums

The **RUNNINGSUM** function sums the value of a measure or an expression with all preceding values that would appear in the same column. Unlike the **MovingSum** function, which limits the number of preceding values such that one value is subtracted from the sum as another is added, the **RUNNINGSUM** function continuously increments the value returned by the function from the top of the report to the bottom until the last value represents the grand total for that column. The only argument required by this function is the name of the measure on which the function is being calculated:

```
MyQuery.QueryItems.Add "RUNNINGSUM (MEASURE)"
```

Pivoting measures to the row orientation directs this function to increment values along a row rather than down a column. Substituting new constant values in Exercise 15 and executing the modified procedure demonstrates this effect:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "4 Week Period"
    Const SecondPivot = 2    'By Columns

    Const Expression = "RunningSum (Units Sold)"
    Const MeasurePivot = 1   'By Rows
```

Like moving averages and moving sums, running sums are often calculated over time, so the **Region** attribute must be replaced with an attribute from the time dimension, **4 Week Period**, and measures must be pivoted such that the function increments values along a row rather than down a column.

The resulting report is displayed in Figure 5-13. That report adds the number of units sold for any brand to corroborate the idea that the value returned by the **RUNNINGSUM** function increases from one four-week period to the next by the volume of sales for each period, giving the total sales to date for that brand at four-week intervals.

Figure 5-13
Running Sum for a Brand-Region Query

Brand	4-Week Period	94/01/01 to 94/01/28	94/01/29 to 94/02/25	94/02/26 to 94/03/25	94/03/26 to 94/04/22
Alden	Units Sold	584	616	748	219
Alden	RunningSum (Units Sold)	584	1200	1948	2167
Barton	Units Sold	425	418	579	159
Barton	RunningSum (Units Sold)	425	843	1422	1581
Delmore	Units Sold	644	520	762	161

Brand	4-Week Period	94/01/01 to 94/01/28	94/01/29 to 94/02/25	94/02/26 to 94/03/25	94/03/26 to 94/04/22
Delmore	RunningSum (Units Sold)	644	1164	1926	2087
Extreme	Units Sold	147	138	195	55
Extreme	RunningSum (Units Sold)	147	285	480	535
Techno	Units Sold	1242	1126	1614	347
Techno	RunningSum (Units Sold)	1242	2368	3982	4329

Top N

The **TOPN** function evaluates the values of a measure or an expression for a specified column, limiting the rows displayed in the entire report to those with the highest or lowest values for that column. For example, this function could limit a query returning brand sales data to three rows, which represent the top three selling brands. The values returned by the function for those brands would simply be sales data.

Since reports often feature multiple columns of data, you must specify the column on which the function should base its selection of the top or bottom rows. If you subdivide brand sales into two columns by region, Northeast and West, the function can limit the query to the top three selling brands in the Northeast region or the top three selling brands in the West region.

Because of this potential ambiguity, the **TOPN** function requires four arguments: the name of the measure or expression to be evaluated for high or low values, the number of rows to include in the report, the column on which to base the function, and a flag, `Asc` or `Desc`, indicating whether to take the highest or lowest values. Setting the `Desc` argument directs the function to choose the lowest rather than the highest values. `Asc` and `Desc` are not constant names substituting for a numeric flag but strings understood directly as arguments by the function. The syntax for including this function as an expression when instantiating a QueryItem is:

```
MyQuery.QueryItems.Add _
    "TOPN (MEASURE, # OF ROWS, COLUMN/ROW INSTANCE, _
        Asc/Desc FLAG)"
```

The `Column/Row Instance` argument identifies the column on which to base the selection of the top or bottom rows by number. Columns are numbered in a report from left to right, beginning with `0`, ignoring subdivisions created for different measures. If no attributes have been pivoted to the column orientation, the value for this argument should be `0`.

Because columns are subdivided by different attribute values, it becomes more difficult to define the basis for choosing the top or bottom rows of data.

Consider the report displayed in Figure 5-14. The top brands can be identified on the basis of numeric data for the Northeast or West region, for either 1995 or 1994. For each column, the two highest values are set in boldface. A **TOPN** function that returns revenues for the top two revenue-grossing brands would display different brands, depending on the column specified by the column/row number argument: Alden and Barton were the top-grossing brands in the Northeast region for 1994; but in 1995, Alden and Lasertech were the top-grossing brands.

You cannot, however, identify the critical column by the name of a single attribute value, such as Northeast, because multiple attributes may subdivide columns. In this example, you must specify whether the function should return the highest values for the Northeast in 1994 or 1995. And while this query subdivides columns by two attributes, a different query might subdivide columns by more attributes.

It is for this reason that you must identify columns by the more flexible convention of index number.

Figure 5-14
Report Subdivided by Different Attribute Values, Measures, with Highest Two Values in Bold for Each Column

Column #:	**0**		**1**		**2**		**3**	
Region	**Northeast**				**West**			
Fiscal Year	1994		1995		1994		1995	
Brand	Units Sold	Gross Revenue	Units Sold	Gross Revenue	Units Sold	Gross Revenue	Unit Sold	Gross Revenue
Alden	899	**3062700**	912	**3014520**	1268	**4268520**	1358	**4494860**
Barton	638	**688660**	676	717500	943	1033420	981	**1050360**
Delmore	861	191750	917	205850	1226	278400	1331	299600
Extreme	212	371000	221	386750	323	565250	326	570500
Lasertech	519	361700	586	**901800**	814	581100	851	588750
NVD	**1349**	218980	1310	220940	**1796**	287530	**1992**	320025
Onetron	442	68525	468	74950	**2600**	99975	654	108750
Suresound	1230	297540	**1318**	325600	1693	**399210**	1771	420090
Techno	**1817**	104024	**1882**	108175	251	145285	**2774**	158247

Since the measure on which to base the **TOPN** function is already explicitly specified as an argument, subdivisions created for different measures do not increment the column / row instance. The column / row instance only increments when a new combination of attribute values groups data by columns. In Figure 5-14, each such combination is shaded differently. A row at the head of the report identifies the column number.

Since the column / row number depends on the order in which different columns appear, reversing the sort on any attribute with the same orientation as measures changes the basis for defining the top or bottom rows. Pivoting the orientation of measures to rows directs the function to limit the query result to a certain number of columns, based on the values appearing in the row specified by the column / row number. The column / row number increments in the same way, beginning with the topmost row and going down.

To deploy this function in a procedure, substitute new constants in Exercise 15:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2    'By Columns

    Const Expression = _
"TOPN (Units Sold, 2, 0, Asc)"
    Const MeasurePivot = 2   'By Columns
```

Executing the modified procedure generates the report displayed as Figure 5-15. As the first value of an attribute organized by columns, Northeast occupies the position of the zeroth column, which the column/row number argument specifies as the basis for calculating the top-selling brands.

Figure 5-15
Top-Two Selling Brands for the Northeast Region

Region	**Northeast**		**West**	
Brand	Units Sold	TOPN (Units Sold, 2, 0, Asc)	Units Sold	TOPN (Units Sold, 2, 0, Asc)
NVD	2719	2719	3788	3788
Techno	3699	3699	5286	5286

After the **TOPN** function evaluates the values of the specified measure, the top or bottom rows are identified and selected, and the function returns the measure values for those rows unchanged. Separately including the measure on which the **TOPN** function is based results in the redundancy seen in Figure 5-15.

Top Percentage

The **TOPPCT** function is nearly identical to the **TOPN** function except for the second argument. The second argument represents the percent by which displayed rows can be less than the highest value. For example, if this function were performed on a column for which the top value was 100, any rows for which the value of that column were 90 or higher would be returned by a **TOPPCT** function set to 10 percent. This would be true even if the value of all rows for that column were between 90 and 100, in which case the **TOPPCT** would essentially have no effect. The second argument is thus a number between 1 and 100 representing a percentage rather than an absolute number.

To deploy this function in a procedure, substitute new constants in Exercise 15:

```
'Declare Constants
    Const FirstAttribute = "Brand"
    Const FirstPivot = 1     'By Rows

    Const SecondAttribute = "Region"
    Const SecondPivot = 2    'By Columns

    Const Expression = _
        "TOPPCT (Units Sold, 50, 0, Asc)"
    Const MeasurePivot = 2   'By Columns
```

Executing this query returns only three rows. The top-selling brand for the specified column **Northeast** is Techno Components, with sales of 3699. Only two other brands enjoyed sales in the Northeast within 50 percent of 3699 units. Both are included in the report shown in Figure 5-16.

Figure 5-16
Brands Within Fifty Percent of the Top-Selling Brand in the Northeast Region

Region	Northeast		West	
Brand	Units Sold	TOPPCT (Units Sold, 50, 0, Asc)	Units Sold	TOPPCT (Units Sold, 50, 0, Asc)
NVD	2719	2719	3788	3788
Suresound	2548	2548	3464	3464
Techno	3699	3699	5286	5286

This function does not limit the number of rows appearing in the report to 50 percent of all rows, in which case the report would have included four or even five rows.

The general syntax for this function is:

```
MyQueryQueryItems.Add "TOPPCT (MEASURE,
% OF ROWS TO DISPLAY, COLUMN/ROW INSTANCE, Asc/Desc FLAG)"
```

Nesting QueryItem Expressions

Any function that performs a calculation on the numeric data represented by a measure can also perform that calculation on the data returned by another function. All of the measure functions in the main MetaCube extension can thus be nested one within another. For example, a query requesting the three brands that have experienced the most dramatic change in sales from one period to the next nests the **ABS_CHANGE** function within the **TOPN** function:

```
MyQuery.QueryItems.Add _
    "TOPN (ABS_CHANGE (Units Sold), 3, 0, Asc)"
```

In this example, MetaCube performs a recursive analysis. For every two columns of raw data, MetaCube first calculates the difference between the two, interpolating a column displaying the result. MetaCube then identifies the brands for which the first, or zeroth, column of absolute change values are largest, only displaying absolute change values for those brands.

Extension Functions as QueryCategory Expressions

The **BUCKET** and **COMPARE** functions embed query capabilities in MetaCube's analysis engine. The **COMPARE** function in particular expands the definition of a QueryCategory to include heterogeneous information, requiring MetaCube to issue multiple queries to the relational database, and subsequently to consolidate the result in a single report.

Figure 5-17
QueryCategory Functions

Function	Description/Example
BUCKET	Sums specific values of a single attribute in user-labeled groups. `MyQuery.QueryCategories.Add "BUCKET _ (Brand, [FirstLabel, list (Techno Components, Suresound)], _ [SecondLabel, [a, f]], [ThirdLabel, other])"`
COMPARE	Retrieves values of multiple attributes. A different filter can be applied to each attribute. `MyQuery.QueryCategories.Add "COMPARE _ (NOSORT [Brand, Video Only], [Product Class, Audio Only])"`

The syntax for each function is explained below. As always, if the instance of the QueryCategory is stored in an object variable, the entire expression must be enclosed in parentheses, as explained in "Declaring MetaCube Object Type Variables" on page 1-14.

Bucket

The **BUCKET** function groups selected values of a single attribute under customized headings, creating a QueryCategory based on that attribute but with entirely different, user-defined groupings. Each grouping is referred to as a bucket. For example, for the **Brand** attribute this function could create a bucket named **My Brands**, consisting of the Alden and Delmore brands, and another bucket named **Other Brand**s, consisting of the remaining six brands. The syntax for this function is embedded as an expression when instantiating a QueryCategory:

```
MyQuery.QueryCategories.Add "BUCKET (Brand, [My Brands, list
(Alden, Delmore)], [Larry's Brands, [ex, su]], [Other Brands,
other])"
```

Each set of brackets defines a bucket, and each bucket defines a row or column in a report. In this example, assuming the QueryCategory has a row orientation, the resulting report features three rows, one showing sales for **My Brands**, another the sales for **Larry's Brands**, and a third the sales of **Other Brands**. The syntax for each bucket is preceded by an argument labelling that bucket with a name. The grammar for specifying the values included within that bucket has the following three possible components:

- A list, with specific attribute values in parentheses: `[BUCKETNAMEA, list (Alden, Extreme)]`
- A letter or set of letters or numbers in brackets defining a range of values: `[BUCKETNAMEB, [a, z]]`.

  You may specify the beginning or end of a letter-defined range with multiple letters, requesting, for example, attribute values beginning with `qu` and ending with `th`. The syntax for letter-defined ranges can take four possible forms:

  - All values that begin with a letter or set of letters and *before*: `[BUCKETNAMEB, [all, n]]`
  - All values that begin with a letter or set of letters and *after*: `[BUCKETNAMEB, [r, all]]`
  - All values that begin with a letter *between* two letters, including those letters: `[BUCKETNAMEB[o,q]]`
  - Simply all values [all, all]

  Reversing the order in which MetaCube sorts attribute values does not affect buckets defined by letter ranges. If values are numeric, numbers can be substituted for letters, with the obvious caveat that numbers must be provided in their entirety as opposed to a stem of the first digit or digits.
- The category *other*, including all attribute values excluded from other buckets in the expression: `[BUCKETNAMEC, other]`

You can include as many or as few buckets of the same or different types as you require.

Buckets defined using letter- or number-ranges and lists can overlap, including the same attribute value in multiple groupings. If the attribute on which that bucket is based is included as a QueryCategory in a report with the bucket pivoted to the same orientation, the attribute value appears twice, once for each bucket. The numeric sum of the rows or columns featuring overlapping buckets always increases, since grouping the same transactional data into different buckets repeats values, increasing the total.

The syntax for instantiating a QueryCategory using a bucket expression that includes all three types of buckets can be generalized:

```
MyQuery.QueryCategories.Add "BUCKET (Attribute Name, _
    [Label1, list ("Value1", "Value2",…)], [Label2, _
    [number/letter, number/letter]], [Label3, OTHER])"
```

Exercise 16 on page 5-47, illustrates the deployment of buckets in a simple procedure.

Compare

The **COMPARE** function includes the values of multiple attributes in a single QueryCategory, applying to each attribute a different filter. For example, a **COMPARE** function could create a QueryCategory representing sales of three brands as well as the sales of two product lines. Two attributes are involved, **Brand** and **Product Line**, and two filters, one limiting the number of brands to only three, the other limiting product lines to only two, resulting in a total of five different groupings.

The resulting report thus features five columns or five rows for that Query-Category, depending on its orientation. To answer this query, MetaCube issues two queries, one for **Brand**, the other for **Product Line**. Bringing information about both product lines and brands to one report enables analysts to compare seemingly dissimilar groupings.

The **COMPARE** function can include any number of attributes, each filtered differently. Moreover, the filters applied to such attributes can limit data by multiple criteria, even including constraints from different dimensions. For example, the filter on **Brand** could also include a constraint on time, limiting the sales data for those three brands to the most recent six months of sales. Although a filter may be comprised of multiple constraints, you cannot apply more than one filter to a single attribute within a comparison unless you actually compare an attribute to itself, such that the attribute is repeated as an argument but each time with a different filter applied.

The **COMPARE** function can be thought of as returning a set of attribute values to a QueryCategory. As such, this function appears as an expression when instantiating a QueryCategory. For each entity included in the comparison, you must specify the name of an attribute followed by the name of the filter to be applied to that attribute. Each entity appears in brackets.

As noted previously, the **COMPARE** function can incorporate any number of entities in a single QueryCategory. For each entity included in a comparison, MetaCube must issue a separate SQL statement, consolidating results within MetaCube's analysis engine.

The example provided below applies the Audio Only filter to **Brand**, comparing sales for that entity to a second entity in which a Computer Only filter is applied to **Product Subclass**:

```
MyQuery.QueryCategories.Add "COMPARE (NOSORT,
[Brand, Audio Only], [Product Subclass, Computer Only])"
```

The first argument indicates whether the values of all entities should be sorted as a group or whether each entity should remain distinct. In our example, the `NOSORT` flag directs the function to sort the two entities separately so that brand names appear first and product subclass names second.

Conversely, the `SORT` flag directs the function to sort all attribute values together, irrespective of the entity to which they belong. Applying an indiscriminate sort to a comparison between brands and product subclasses would, for example, result in a report in which brand names like Lasertech appear beside product subclass names like Laser Disc Players.

Regardless of whether the entities are sorted separately or together, the actual direction of the sort can be set by assigning a value to the QueryCategory**'s SortDirection** property, as documented in Figure 8-7 on page 8-25.

The syntax for a **COMPARE** function can be generalized as follows:

```
MyQuery.QueryCategories.Add "COMPARE, (SORT/NOSORT,
[Attribute Name, Filter Name], [Attribute Name,
Filter Name],…)"
```

If you substitute the word *ALL* for the filter name, no filter is applied to the entity.

Exercise 16 illustrates the deployment of both **BUCKET** and **COMPARE** functions in a procedure. The **BUCKET** function groups brands by their hypothetical brand managers, Brendan and Glenn. The **COMPARE** function retrieves brand sales for a region, Northeast, and a city within that region, New York. A **PCT_PREV** function calculates the fraction of regional sales that can be attributed to New York for both brand managers.

Exercise 16: Buckets and Comparisons

```
77  Option Explicit
78
79  'MetaCube API Exercise 16: Buckets and Comparisons
80
81   Sub MetaCube_API()
82
83  'Declare Variables
84      Dim MyMetabase As Object, MyQuery As Object, _
85          MyMetaCube As Object, MyData As Variant
86      Const OrientationColumn = 2
87
88  'Excel Variables
89      Dim ReportRange As Range, _
90           MyListBox As ListBox
91
92  'Connect
93      Set MyMetabase = CreateObject("Metabase")
94
95      'Identify an ODBC Data Source
96       Let MyMetabase.ConnectString = "demo41"
97
98       'Specify a set of metadata
99       Let MyMetabase.Name = "MetaCube Demo"
100
101      'Specify a schema name
102      Let MyMetabase.MetaSchema = "metacube."
103
104      'Specify a Database Type - 1 = Oracle  5 = Informix
105      Let MyMetabase.ConnectDatabase = 5
106
107      'Specify login to database
108      Let MyMetabase.Login = "metademo"
109      'Passes value to database on connection
110
111      'Specify database password
112      Let MyMetabase.Password = "md$$pwd"
113
114     'Enable this Extension if necessary
115     'MyMetabase.extensions.Add _
116     '     "c:\metacube\mcplgmn.mcx"
117     MyMetabase.Connect
118
119 'Define Query
120     Set MyQuery = MyMetabase.Queries.Add("Untitled1")
121     MyQuery.QueryCategories.Add _
122         "BUCKET (Brand,[Glenn's Sales,
    List(Alden,Barton,Extreme)], [Brendan's Sales,
```

```
    List(Onetron,NVD,Lasertech)])"

    MyQuery.QueryCategories.Add _
        "COMPARE (NOSORT, [Region, Northeast],[City, New
York])"

    MyQuery.QueryCategories.Item(1).Orientation = _
        OrientationColumn
    MyQuery.QueryItems.Add "Units Sold"

    MyQuery.QueryItems.Add _
    "PCT_PREV (Units Sold)"

'Get Results
    Worksheets.Item("Query Report").Activate
    Cells.Select
    Selection.ClearContents

    Set MyMetaCube = MyQuery.MetaCubes.Add("Data")
    Let MyData = MyMetaCube.ToVBArray
    Set ReportRange = ActiveSheet.Range _
        (ActiveSheet.Cells(1, 1), _
        ActiveSheet.Cells _
        (MyMetaCube.Rows, MyMetaCube.Columns))
    Let ReportRange.Value = MyData
    ReportRange.EntireColumn.AutoFit

End Sub
```

Executing this procedure generates the report displayed in Figure 5-18, in which brands are grouped into two buckets, labeled Brendan and Glenn, and sales of those brands are evaluated for the Northeast region and the city of New York. A **PCT_PREV** function compares the two columns of data, indicating the extent to which New York contributes to the sales in the Northeast region of the brands managed by Brendan and Glenn.

Figure 5-18
Report Generated by Exercise 16

COMPARE (NOSORT, [Region, Northeast], [City, New York])	**Northeast**	**New York**	**New York**
BUCKET (Brand, [Glenn's Sales, List(Alden, Barton, Extreme)], [Brendan Sales, List(Onetron, NVD,Lasertech])	Units Sold	Units Sold	PCT_PREV (Units Sold)
Glenn's Sales	3558	1393	39.151
Brendan's Sales	4734	1816	38.36

Chapter

6

The FactTables Class of Objects and Related Collections

In This Chapter

This chapter introduces the **FactTables** class of objects and all the collections either directly or indirectly belonging to objects of this class. The **FactTables** object class, and the collections belonging directly or indirectly to that class, enable you to develop procedures that create, edit, or access MetaCube's metadata.

The FactTables Class of Objects

A fact table stores all transaction-level data in the data warehouse and sits at the center of a star or snowflake model. The FactTable object describes the physical location of this table as well as the dimensions to which it joins, its size, the measures it contains, and other information.

The FactTables Collection's Add Method

To instantiate a FactTable object, you must identify the parent metabase that owns the collection of FactTable objects to which this object will belong as well as the following arguments: the name of the newly instantiated object, the name of the database fact table, and the schema/location of that table. You present these arguments in the format: `MyMetabase.FactTables.Add` *`Name, Schema, Table`*.

For example, to instantiate a FactTable object named **Marketing Data Source** based on a table in the **METADEMO** schema named **MARKETING_FACT**, you would execute the following command:

```
MyMetabase.FactTables.Add "Marketing Data", _
"METADEMO", "MARKETING_FACT"
```

The arguments of the collection's **Add** method correspond to FactTable properties discussed below.

FactTables Properties

Figure 6-1 summarizes the properties of the **FactTables** object class.

Figure 6-1
FactTables Class of Objects: Properties

Property	Description/Example
Cost	Long integer. The performance cost of accessing the fact table, roughly correlated to the size of the table. MetaCube identifies the optimal table from which to return a result by assessing cost. Defaults to the highest possible number. `MyFactTable.Cost = 30000`
DefaultCost	Long integer. Read-only. The default cost of the corresponding table. `c=MyFactTable.DefaultCost`
DefaultRow-Count	Long integer. Read-only. The default row count of the corresponding table. `r=MyFactTable.DefaultRowCount`
IconBitmap	String. Warehouse Manager converts the icon associated with a fact table from a bitmap to string information and stores this string in the database. This value's property is thus the string representation of the icon. `MsgBox MyFactTable.IconBitmap`
IconName	String. Name of original bitmap file for storing the icon that represents this dimension. No default. `MyFactTable.IconName = "CASH.ICO"`
MeasureNames	ValueList of names for all of the fact table's measures. Arguments: Display type constants to display the items validated for queries, filters, or both. See Figure 4-2 on page 4-8. `MsgBox MyFactTable.MeasureNames(2)`

(1 of 3)

Property	Description/Example
MeasuresFirst	Boolean. This flag can be used to determine the order in which to display measure and dimension names to an end user. ■ `False` indicates that dimensions should display first, then measures. This is the default. ■ `True` indicates that measures should display first, then dimensions. `MyFactTable.MeasuresFirst=False`
Name	String. The name of the fact table. Default property. `MsgBox MyFactTable.Name`
Parent	Object. The Metabase object owning the collection to which this fact table belongs. `MsgBox MyFactTable.Parent.Name`
Schema	String. The schema/location wherein the fact table physically resides. `MyFactTable.Schema = "METADEMO"`
Table	String. The name of the database table itself. `MyFactTable.Table = "SALES_TRANSACTIONS"`
TableSize	Long. Stores the precise number of rows in the table, irrespective of other performance issues such as indexing or partitioning. The administrator must enter a value for this property so that MetaCube can compare the size of sample tables to the original fact table and thereby derive the margin of error attributable to such samples. `MyFactTable.TableSize = 12102`

(2 of 3)

Property	Description/Example
ValidFlag	Boolean. A `True` value indicates the fact table is valid for querying. Default upon instantiation is `False`. `MyFactTable.ValidFlag = True`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a fact table is one of the following: ■ Completely valid ■ Invalid because at least one Aggregate, AggregateMeasure, DimensionMapping, Measure, or Sample object owned by the FactTable object is invalid ■ Invalid because the **FactTable** object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyFactTable.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the FactTable object's metadata. This will not include errors in the metadata for other objects belonging to the FactTable object. `MsgBox FactTable.VerifyResults.TabbedValues`

(3 of 3)

FactTables Methods

Aside from the standard **Verify** method, the **FactTables**' only method is the **WriteIcon** method, which converts the string value of the **IconBitMap** property to a standard icon file on the client. This method allows you to specify the directory in which to create the icon file, with the MetaCube directory as the default:

```
MyFactTable.WriteIcon "C:\METACUBE"
```

This method parallels the structure of the Dimensions object class' **WriteIcon** method, discussed in "Dimensions Methods" on page 4-7.

FactTables Collections

A FactTable object's collections include objects representing all of the tables and columns from which a query could possibly retrieve data, including the dimension tables to which the fact table joins to consolidate transactional information, the aggregate tables storing summarizations of those transactions, and the columns storing the numerical measures.

Figure 6-2 explains each of the FactTable object's collections.

Figure 6-2
FactTables Class of Objects: Collections

Collection	Description/Example
Aggregates	A collection of objects that describe aggregate tables. Aggregate tables summarize transactions stored in the fact table to deliver better query performance. `MsgBox MyFactTable.Aggregates.Names`
Dimension-Mappings	For each dimension to which a particular fact table joins, there must exist a DimensionMapping object describing the join itself as well as how to display the dimension to users querying that fact table. Because a dimension can join to more than one fact table and because for each fact table an application may present the dimension in a different way, you can define the relationship between a fact table and a dimension separately through the objects in this collection. A fact table's DimensionMapping objects should correspond exactly to a fact table's Dimension objects. `MsgBox MyFactTable.DimensionMappings.Names`

(1 of 2)

Collection	Description/Example
Dimensions	A subset of the DSS System's library of available dimensions, consisting of those dimensions to which the fact table joins: that is, a subset of the Dimension objects in a collection owned by a Metabase object. Deleting a dimension from this collection only signifies that the dimension cannot join to the fact table owning the collection. Other fact tables can continue to join to this dimension. However, any changes made to a dimension object within this collection are registered for the entire DSS System. `MsgBox MyFactTable.Dimensions.Names`
Measures	A collection of numeric, additive metrics stored in, or calculated from, columns in the fact table. `MsgBox MyFactTable.Measures.Names`
Samples	Consists of objects describing sample tables. Sample tables store a statistically significant, evenly distributed set of records replicated from the fact table. Sophisticated statistical algorithms enable MetaCube to extrapolate query results within a prescribed range of accuracy from sample tables. Since sample tables are a fraction of the original fact table's size, such tables offer better performance. `MsgBox MyFactTable.Samples.Names`

(2 of 2)

The Aggregates Class of Objects

This object class describes aggregate tables. Aggregate tables improve query performance by storing summary-level data. It is only a slight simplification to understand each aggregate table as a repository for a certain query result. If a user requests that result or information that can be derived from that result, MetaCube can satisfy his or her request more quickly by retrieving information from the aggregate table.

Of course, a query requesting information at any level of summarization can always retrieve transaction level information from the fact table, consolidating the detail into larger groupings by joining to dimension tables. But scanning and joining large tables poses intractable performance problems. Aggregates enable certain queries to bypass large fact tables and sometimes dimension tables to reduce the number of rows the database must process. In an intelligently aggregated data warehouse, only queries requesting detail-level data require fact table processing.

Aggregate tables summarize transactions in the fact table by a complex set of high-level elements in a dimensional hierarchy. For each fact table, there should exist a set of aggregate tables to improve performance for queries against that fact table. Although aggregates improve performance, you need not expose a view of aggregates to users of a MetaCube query application, as MetaCube automatically and transparently routes each query to the optimal aggregate table, if one exists. For each fact table, there should exist a set of aggregate tables to improve performance for queries ostensibly against that fact table.

Each aggregate table is thus associated with both the fact table that it summarizes and its corresponding **Aggregate** object existing within a collection owned by a **FactTable** object. Since aggregates are always calculated directly from a fact table, an **Aggregate** object cannot exist outside of a collection owned by a fact table.

The Aggregates Collection's Add Method

Instantiating an Aggregate object to describe or create a new summary table in the relational database requires you to specify three arguments: the name of the object, the name of the schema/location storing the table represented by that object, and the name of the table itself. For example, to describe a table named **SALES_AGG1** in the **METADEMO** schema, we could instantiate an Aggregate object named **Sales**:

```
MyFactTable.Aggregates.Add "Sales", "METADEMO", "SALES_AGG1"
```

The new object will belong to **MyFactTable**'s collection of Aggregate objects.

Aggregates Properties

An Aggregate object's properties can either describe an existing aggregate table, or enable MetaCube to generate the SQL statements to create a new aggregate table. Whereas the values of Dimension and FactTable objects merely describe database tables, the values of an Aggregate object's properties constitute a blueprint for building tables.

In truth, the values of all three object classes' properties populate MetaCube's metadata tables, which traditionally describe the data model. Unlike other object classes, however, the Aggregates object class can generate SQL on the basis of the metadata, in effect reverse-engineering a database table from the metadata description. MetaCube can populate the new table by querying existing fact and dimension tables. If the underlying aggregate table already exists, the values of the Aggregate object's properties as well as its collections depend on that table's physical characteristics. If the table does not yet exist, the characteristics of that table depend on the values of the Aggregate object's properties and on its collections.

MetaCube Aggregator, MetaCube's server-side agent for aggregate construction and maintenance, can execute the SQL stored in the **CreateStatement** and **TableOptions** properties. Aggregate object properties such as the **SchemaPassword** property exist to automate the process of actually building the aggregate through Aggregator. You can also execute the SQL MetaCube generates from any other database development environment.

Figure 6-3 summarizes the properties of the **Aggregates** class of objects.

Figure 6-3
Aggregates Class of Objects: Properties

Property	Description/Example
Cost	Long integer. The performance cost of accessing the aggregate table, roughly correlated to the number of rows in the table. MetaCube identifies the optimal table from which to return a result by assessing costs. The values of an aggregate's cost and a fact table's cost should be similarly scaled, so that, for example, the cost of a fact table containing twice as many rows as an aggregate will be twice as high as the aggregate's cost. Defaults to the highest possible number. `MyAggregate.Cost = 30000`
DefaultCost	Long integer. Read-only. The default cost of the corresponding table. `c=MyAggregate.DefaultCost`
DefaultRow-Count	Long integer. Read-only. The default row count of the corresponding table. `r=MyAggregate.DefaultRow Count`
Create-Statement	String. For any completely defined Aggregate object, MetaCube can generate SQL to create the physical table that the object ostensibly describes, storing the CREATE statement as a value of this property. In other words, you can instantiate and define an Aggregate object and its collections when the actual aggregate table does not exist and thereby generate the SQL to create the table. MetaCube Aggregator, a server-side agent, can execute this SQL when prompted to do so. The **GenSQL** method generates the SQL stored by this property. `MsgBox MyAggregate.CreateStatement`

(1 of 3)

Property	Description/Example
Filter	Filter Object. This read-only property identifies a Filter object, which is a collection of constraints by which the aggregate table is partitioned. Each constraint is instantiated as a FilterElement object. The Filter object owned by an aggregate is identical to the Filter object owned by a query, although it cannot be opened or saved. Other properties and methods that normally apply to a Filter object are invalid. Just as you can filter a query, effectively placing a WHERE clause in the SQL retrieving your query result, you can place a constraint within the SQL generated to build and populate an aggregate table. Defaults to empty. `MsgBox MyAggregate.Filter.Name`
LastUpdate	Variant, date. Indicating the date of the aggregate's most recent update, for maintenance purposes. `MsgBox MyAggregate.LastUpdate`
Name	String. The name of the Aggregate object. Default property. `MsgBox MyAggregate.Name`
Parent	Object. The FactTable object owning the collection to which the aggregate belongs. `MsgBox MyAggregate.Parent.Name`
Schema	String. The name of the schema storing the aggregate table. `MyAggregate.Schema = "METADEMO"`
Schema-Password	String. A password string for the database schema storing the aggregate, stored in an encrypted format on the database. MetaCube Aggregator requires this password to execute the SQL building the aggregate in that schema. `MyAggregate.SchemaPassword = "WELCOME"`
Table	String. The name of the aggregate table. `MyAggregate.Table = "SALES_AGG9"`
TableOptions	String. Beyond the commands represented by the **Create-Statement** property, this property stores any database-specific or custom SQL statements necessary to build the aggregate. MetaCube Aggregator executes the **TableOptions** commands together with the CREATE statement. `MsgBox MyAggregate.TableOptions`

(2 of 3)

Property	Description/Example
ValidFlag	Boolean. Indicates whether the aggregate is currently valid for queries; defaults to `False`. `MyAggregate.ValidFlag = False`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for an aggregate table is one of the following: ■ Completely valid ■ Invalid because at least one AggregateMeasure or Aggregate-Group object owned by the Aggregate object is invalid ■ Invalid because the **Aggregate** object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyAggregate.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the Aggregate object's metadata. This will not include errors in the metadata for other objects belonging to the Aggregate object. `MsgBox MyAggregate.VerifyResults.TabbedValues`

(3 of 3)

Aggregates Methods

Aside from the standard **Verify** method, the **Aggregates** class of object features only one method, **GenSQL**, which prompts MetaCube to generate SQL from the metadata description of a new Aggregate object. Because the properties and collections of the Aggregate object already specify all of the parameters for creating the database table, this method requires no arguments. **GenSQL** returns the SQL as a string value:

```
MyAggSQL$ = MyAggregate.GenSQL
```

Aggregates Collections

The Aggregate object is, in large part, defined by several collections of component objects. Figure 6-4 summarizes these collections.

Figure 6-4
Aggregates Class of Objects: Collections

Collection	Description/Example

Aggregate-Grants	A collection of objects representing SQL GRANT statements, which are used for conferring privileges to view aggregate tables.
	`MyAggregate.AggregateGrants.Add _`
	`"GRANT SELECT ON sales_agg1 to METADEMO"`

Aggregate-Groups	Each AggregateGroup object identifies a DimensionElement or Attribute object by which data is summarized in the aggregate table. This attribute or dimension element must be included in one of the dimensions to which the fact table joins. The objects in the **AggregateGroups** collection also identify the column in the aggregate table storing that dimension element or attribute's values.
	`MyAggregate.AggregateGroups.Add _`
	`MyDimEl, "BRAND_CODE"`

Aggregate-Indexes	A collection of objects storing indexes for a new aggregate table. Aggregator automatically executes the SQL statement creating the index when it builds the aggregate. When instantiating the AggregateIndex object, include the SQL statement as an argument.
	`MyAggregate.AggregateIndexes.Add _`
	`"CREATE UNIQUE INDEX myindex ON...", _`
	`"METACUBE.", ""`

Aggregate-Measures	Each AggregateMeasure object identifies one of the fact table's Measure objects, thereby including that measure in the aggregate table as well. When instantiating this object, you must also identify the column in the aggregate table storing that measure.
	`MyAggregate.AggregateMeasures.Add _`
	`MyMeasure, "UNITS_SOLD"`

The AggregateGrants Class of Objects

This object class allows you to define the grants for a new aggregate, which MetaCube Aggregator executes when creating the aggregate table. Aggregate objects that describe existing tables need not include any AggregateGrant objects in this collection: the necessary grants probably already exist.

Aside from the **Parent** property, which identifies the Aggregate object that owns the AggregateGrant object's collection, the only property of the AggregateGrant object is the **GrantStatement** property. The string value of this property, which you specify as an argument when instantiating an AggregateGrant, is simply the GRANT SQL statement, which MetaCube stores in the metadata tables for MetaCube Aggregator to execute when building the aggregate. An example of this syntax can be found in Figure 6-4.

The AggregateGrant object does not feature any collections or methods. Specifically, MetaCube does not include a method for generating GRANT SQL statements. You must formulate them yourself.

The AggregateGroups Class of Objects

Each AggregateGroup object describes one of the dimension elements or attributes by which you summarize data in an aggregate table. Whereas a fact table may define a sales transaction in terms of base dimension elements such as product code or store code, an aggregate table groups sales by brand or by region. As the number of dimensions increases, identifying the optimal level of detail at which to summarize transactions in aggregate tables can become difficult. MetaCube Warehouse Optimizer can analyze your data warehouse to recommend the most effective set of aggregates to build.

Aside from the standard **Parent, Verified,** and **VerifyResults** properties, this object class features only two other properties, no collections, and only one method, the **Verify** method. The verification properties and methods evaluate the validity of the metadata defined by an object's other properties, as explained above.

The first property, **Category,** refers to either an Attribute or Dimension-Element object that defines the level of detail for a particular dimension by which metrics are grouped. You must include this argument when instantiating an **AggregateGroup** object:

```
MyAggregate.AggregateGroups.Add MyFactTable. _
Dimensions.Item(0).DimensionElements.Item(1), "BRAND_CODE"
```

The second argument in this command, a value of the **Column** property, identifies the column in the aggregate table in which the actual dimension element or attribute values are stored.

An aggregate can group transactions by values of any attribute or dimension element that belongs to a dimension joined to the fact table. However, aggregates that summarize information by dimension element values are much more flexible and powerful, because they can join to dimension tables to process any query requesting a level of summarization equal to or greater than the level stored in the aggregate itself. Since attribute values do not represent a key to any other table, aggregates built by attribute can only process queries requesting data grouped by that particular attribute's values.

You must instantiate an AggregateGroup object for each dimension element or attribute included in the aggregate table, regardless of whether that table already exists or remains to be built. MetaCube determines an aggregate's level of summarization and its suitability for processing a given query by evaluating the properties of that aggregate's AggregateGroup objects.

The AggregateIndexes Class of Objects

The AggregateIndex object stores any indexes you want to place on a new aggregate table. Although MetaCube cannot generate indexes, MetaCube Aggregator executes any SQL statements that are stored in the AggregateIndex object's **IndexStatement** property when building the aggregate table. You need not instantiate AggregateIndex objects for existing aggregate tables, since MetaCube invokes this object only when building new aggregates.

To instantiate an AggregateIndex object you must include the SQL statement creating the index as an argument to the **Add** method:

```
MyAggregate.AggregateIndexes.Add _
"CREATE UNIQUE INDEX myindex ON table(column)"
```

In addition to the **IndexStatement** property, the AggregateIndex object includes properties for storing the name of the schema owning the index, as well as any database-specific or custom SQL statements associated with the index.

Figure 6-5 summarizes the **AggregateIndex** object's properties.

Figure 6-5
AggregateIndexes Class of Objects: Properties

Property	Description/Example
IndexOptions	String. Stores custom SQL parameters for creating an index. `MsgBox MyAggregateIndex.IndexOptions`
IndexSchema	String. Identifies the schema that owns the index. `MyAggregateIndex.IndexSchema = "METADEMO"`
IndexStatement	String. The actual SQL statement used to create the index. Default property. See example above.
Parent	Object. Aggregate object. `MsgBox MyAggregateIndex.Parent.Name`

The AggregateIndex object does not feature any methods or collections.

The AggregateMeasures Class of Objects

The AggregateMeasure object identifies a measure either included in an existing aggregate table or that will be included when a new aggregate is built. Together, an Aggregate object's collection of AggregateMeasure objects define the measures stored in an aggregate table. To include calculated measures in the aggregate table, you need only include the measures on which the calculation for that measure is based.

Regardless of whether you are describing an existing aggregate or a new aggregate, you must instantiate an AggregateMeasure for each measure in the aggregate, because MetaCube evaluates the properties of the Aggregate-Measure object when generating SQL for a query.

An aggregate can only include measures already stored at the transactional level in the fact table. If, for example, a fact table does not store the revenues generated by a transaction, the aggregate table cannot store the revenues generated by a group of transactions.

To identify the measures included in an aggregate table, you must specify the Measure object and the name of the column in the aggregate in which to store the measure identified by the Measure object. Both the Measure object and the column name appear as arguments in the AggregateMeasure collection's **Add** method:

```
MyAggregate.AggregateMeasures.Add _
MyMetabase.FactTables.Item(0).Measures.Item(2), "SALES"
```

These arguments correspond to the **Measure** and **Column** properties, respectively. The **Measure** property identifies a Measure object within the FactTable object's collection of Measure objects. The **Column** property identifies as a string the name of the column that stores that measure's values in the aggregate table. Once you have instantiated an AggregateMeasure object, you can edit the values of either property to reflect changes to the data model:

```
MyAggregateMeasure.Measure = MyFactTable.Measures.Item(1)
MyAggregateMeasure.Column = "GROSS_REVENUES"
```

MetaCube currently only supports aggregates that additively summarize or group information. For example, if you build an aggregate summarizing the measure **Sales** by the aggregate group **Brand Code**, the aggregate totals sales for each brand.

Figure 6-6 summarizes the properties of the AggregateMeasure object.

Figure 6-6
AggregateMeasures Class of Objects: Properties

Property	Description/Example
Column	String. Stores the name of the aggregate table column storing the measure's values. `MyAggregateMeasure.Column = "GROSS_REVENUES"`
Function	String. Identifies the type of summarization to perform on the measure. Defaults to `SUM`, but `COUNT`, `MIN`, and `MAX` functions are also supported. `MyAggregateMeasure.Function = "SUM"`

Measure	Object. Identifies a Measure object from the **FactTables** collection for inclusion in the aggregate. Default property.
	`MyAggregateMeasure.Measure = _`
	`MyFactTable.Measures.Item(1)`

Parent	Object. Aggregate object. `MsgBox MyAggregateMeasure.Parent.Name`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for an aggregate measure is either: ■ Completely valid ■ Invalid because the AggregateMeasure object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyAggregateMeasure.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the **AggregateMeasure** object's metadata. `MsgBox MyAggregateMeasure.VerifyResults.TabbedValues`

Aside from the standard **Verify** method, **AggregateMeasure** objects do not feature any methods and do not own any collections.

The DimensionMappings Class of Objects

The **DimensionMappings** class of objects describes how a particular fact table joins to a dimension table and how to display that dimension table in interfaces. Whereas the Dimension object itself defines a dimension generally, the DimensionMapping object defines a dimension's particular relationship to a fact table.

For each Dimension object in a FactTable object's collection, there must exist a corresponding DimensionMapping object describing the relationship between the two. By describing the relationship between dimensions and fact tables in a separate object, MetaCube enables you to join a dimension to two different fact tables, even though the dimension may join to a different column in each fact table or it should be displayed in a different position for each fact table.

For example, since the relationships between days, weeks, months, and year do not vary from one type of data to another, you might want to join a Time dimension table to both a sales and a marketing fact table. The Dimension object defines the Time dimension, but the DimensionMapping object defines how that dimension joins to each fact table.

When a user chooses to query one of the two fact tables, a property of the DimensionMapping object can determine how to display the dimension in the ensuing query interface. Explorer prompts a user to make just such a choice in the Choose Data Source window, in which each data source corresponds to a different fact table and its associated dimensions.

DimensionMappings Properties

DimensionMapping objects underpin Warehouse Manager's fact table dimensions. The properties of a DimensionMapping object parallel the fields of Warehouse Manager's fact table dimension frame. Figure 6-7 summarizes the DimensionMapping object's properties.

Figure 6-7
DimensionMappings Class of Objects: Properties

Property	Description/Example

Dimension	Object. Identifies an existing dimension object to which the other properties of the DimensionMapping object apply. Default property.
	`My MyDimensionMapping.Dimension = _`
	`MyFactTable.Dimensions.Item(0)`

FactTable-Column	String. Identifies the column in the fact table to which the dimension joins, typically via a base dimension element. `MyDimensionMapping.FactTableColumn = "STORE_CODE"`
Parent	Object. The FactTable object. `MsgBox MyDimensionMapping.Parent.Name`

ScreenOrder	Integer. Determines the order in which Explorer displays the specified dimension for a given data source / fact table. Other MetaCube query applications can also retrieve this integer value to determine the order in which their interfaces display dimensions for a given fact table.
	`ActiveSheet.ListBoxes.Add _`
	`((MyDimensionMapping.ScreenOrder * 80), 50, 70, 100)`

Verified	This property stores a long value returned by the **Verify** method indicating that the metadata represented by a DimensionMapping object is either: ■ Completely valid ■ Invalid because the DimensionMapping object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyDimensionMapping.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the DimensionMapping object's metadata. `MsgBox MyDimensionMapping.VerifyResults.TabbedValues`

Aside from the standard **Verify** method, the DimensionMapping object does not feature any methods, nor does it own any collections.

The Dimensions Class of Objects, as Owned by a FactTable Object

The collection of Dimension objects owned by a particular FactTable object is a subset of the collection owned by a Metabase object and consists of only those dimensions within the DSS System to which the underlying fact table joins. The **DimensionMappings** collection of objects determines which Dimension objects are included in this collection. The DSS System identified by a Metabase object thus includes a library of dimensions, all or only some of which may join to one of the fact tables within that dimension.

All of the properties, methods, and collections related to the **Dimensions** class of objects apply to the object regardless of whether it is identified as a member of a collection owned by a FactTable object or by a Metabase object. However, deleting a DimensionMapping object from a collection owned by a FactTable object only signifies that a particular fact table does not join to that dimension, whereas deleting a Dimension object from a Metabase object's collection of Dimension objects collection deletes the Dimension object generally.

For a full discussion of the Dimension object, its properties, methods, and collections see "The Dimensions Class of Objects" on page 4-3.

The Measures Class of Objects

The **Measure** object represents a type of additive, numerical data stored in the fact table and its aggregates. Examples of a measure in the demonstration data warehouse include **Units Sold**, and **Gross Revenues**. Each **Measure** object corresponds to a column in the fact table storing values of that measure for each transaction or to a calculation based on other Measure objects representing columns in the fact table.

The Measures Collection's Add Methods

To instantiate a standard Measure object, you must specify the name of the measure, and its definition:

```
MyFactTable.Measures.Add _
"Units Sold", "SUM(COLUMN('UNITS_SOLD'))"
```

The first argument identifies the name of the object; the second provides the name of the column in the fact table storing that measure. Once instantiated, this object remains in memory and is saved as metadata only when the Metabase object saves any changes made to the entire DSS System. Both arguments correspond to properties of the Measure object, as explained in the next section.

A separate instantiation method, **AddUserMeasure**, creates a user-defined Measure object, which is available only to the author, as identified by his or her login, and which can be deployed as needed. This method, which requires the same arguments as the standard **Add** method, creates a Measure object that is otherwise indistinguishable from a standard Measure object:

```
MyFactTable.Measures.AddUserMeasure _
"My Units Sold", "SUM(COLUMN('UNITS_SOLD'))"
```

The application need not save the entire DSS System to the database to register the measure. A special method, **SaveUserMeasure**, performs this task:

```
MyFactTable.Measures.Item "My Units Sold" _.SaveUserMeasure
```

Aside from the standard **Verify** method, the **Measures** class of objects does not feature any other methods, nor does it own any collections.

Measures Properties

The properties of a Measure object specify a column in the fact table storing that measure's values or store a formula based on columns in the fact table. These properties also determine which measures are displayed in query and filter interfaces, their order of appearance, and their format. Figure 6-8 summarizes the Measure object's properties.

Figure 6-8
Measures Class of Objects: Properties

Property	Description/Example
BalloonHelp	String. A brief explanation of the measure's significance to end users. Used in balloon help messages such as those available in Explorer. `MyMeasure.BalloonHelp = "The sales metric!"`
Calculated	Boolean, read-only. Stores a true value if the measure is derived from a formula, false if the measure directly accesses data stored in columns of fact and aggregate tables. `MsgBox MyMeasure.Calculated`

Constraint	FilterElement Object. Identifies a constraint to apply against the values of a calculated measure to preclude returning undesirable records, such as negative numbers or zero. For a discussion of measure constraints, see the *MetaCube Warehouse Manager's Guide*. FilterElement objects are discussed in "FilterElements Class of Objects: Properties" on page 8-41.
	`MyMeasure.Constraint.FilterElements.Add _`
	`"Units Sold", ">", "0"`

Definition	String. The definition of this measure, which can identify a column in the database storing a measure or define a formula based on other measures. To identify a measure stored in a column, follow the example provided on the previous page. Each term within a formula must be preceded by the syntax `SUM`, `MIN`, `MAX`, `COUNT` or `AVG` with the argument of that function in parentheses. Each measure is identified by the syntax `FACT`, followed by the name of the measure in single quotes and parentheses. Accepted operators are +, -, /, and *. See the *MetaCube Warehouse Manager's Guide* for more details.
	`MyMeasure.Definition = _`
	`"FACT('Gross Revenue')-FACT('{Profit')"`

DisplayStyle	Long. Indicates whether the measure is valid for display in a query interface, a filter interface, or both. Defaults to both. For a listing of numeric values and their significance as arguments, see the **DisplayStyle** constants in Figure 4-2 on page 4-8. `MyMeasure.DisplayStyle = 2`

(1 of 3)

Property	Description/Example
FormatString	String. Identifies the default format of the measure. The report application or control interprets the contents of the string, and the syntax varies accordingly. The example given here conforms to Microsoft Excel syntax, which is also understood by Explorer's reporting controls. For more information on this topic, see "The FormatString and FormatStrings Properties: An Overview" on page 8-27. `MyMeasure.FormatString = "#,##0.00"`
Name	String. The name of the Measure object. The default property. `MyMeasure.Name = "Net Profit"`
Parent	Object. The FactTable object owning the collection to which the Measure object belongs. `MsgBox MyMeasure.Parent.Name`
ScreenOrder	Long. Stores a value for ordering the appearance of measures in a list box or any other interface control. `MyMeasure.ScreenOrder = 2`
UserMeasure	Boolean, read-only. Stores a true value if the Measure object is user-defined, false otherwise. `MsgBox MyMeasure.UserMeasure`

(2 of 3)

Property	Description/Example
Valid	Boolean. A true value indicates the syntax of the measure definition is valid. Read-only. `MsgBox MyMeasure.Valid`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a measure is either: ■ Completely valid ■ Invalid because the Measure object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox MyMeasure.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the Measure object's metadata. `MsgBox MyMeasure.VerifyResults.TabbedValues`

(3 of 3)

Exercise 17 illustrates how to define and deploy a user-defined measure.

Exercise 17: User-Defined Measures

```
Option Explicit

'MetaCube API Exercise 17: User-Defined Measures


 Sub MetaCube_API()

'Declare Variables and Constants
Dim MyMetabase As Object, MyQuery As Object, _
    MyMeasure As Object, MyMetaCube As Object, _
    MyData As Variant, ReportRange As Range

'Login
Set MyMetabase = CreateObject("Metabase")

'Identify an ODBC Data Source
Let MyMetabase.ConnectString = "Metademo"

'Specify a set of metadata
Let MyMetabase.Name = "MetaCube Demo"

'Specify login to database
Let MyMetabase.Login = "metademo"
'Passes value to database on connection

'Specify database password
Let MyMetabase.Password = "Metademo"

MyMetabase.Connect

'Create User-Defined Measure
Set MyMeasure = MyMetabase.FactTables _
    .Item("Sales Transactions").Measures _
    .AddUserMeasure("Sales by Half", _
    "FACT('Units Sold')/2")
MyMeasure.SaveUserMeasure
MsgBox MyMeasure.Calculated
MsgBox MyMeasure.UserMeasure

'Define Query
Set MyQuery = MyMetabase.Queries.Add("Untitled1")
MyQuery.QueryCategories.Add "Brand"
MyQuery.QueryItems.Add "Units Sold"
MyQuery.QueryItems.Add "Sales by Half"

'Build Report
Set MyMetaCube = MyQuery.MetaCubes.Add("Report1")
```

```
48  Worksheets.Item("User Measures").Activate
49  Cells.Select
50  Selection.ClearContents
51
52
53  'Transform Data in Cube into VB Array
54  Let MyData = MyMetaCube.ToVBArray
55
56  'Import Data into Excel Spreadsheet
57  Set ReportRange = _
58      ActiveSheet.Range _
59      (ActiveSheet.Cells(1, 1), _
60      ActiveSheet.Cells _
61      (MyMetaCube.Rows, MyMetaCube.Columns))
62  Let ReportRange.Value = MyData
63  ReportRange.EntireColumn.AutoFit  'Sizes columns
64
65  End Sub
```

Explanation of Exercise 17

This exercise requires you to name a spreadsheet **User Measures** prior to execution.

This exercise instantiates a Measure object as a private, user-defined object, available for immediate deployment in a query. In lines 8 through 11 declare a set of object variables to store the Metabase, Query, and MetaCube objects necessary to define and to execute a query. You also declare the object variable **MyMeasure**, which stores the user-defined Measure object, and a set of variables for storing data in a Visual Basic array and subsequently displaying that data in a range of spreadsheet cells.

Lines 13 through 29 establish a multidimensional interface to the relational database, opening the DSS System identified in the **Demo** configuration.

Once connected, you can define a new measure in one of two ways. Instantiating a Measure object using the standard **Add** method creates a measure available to all users but first requires you to save the Metabase object. The **AddUserMeasure** method of instantiation, shown on lines 32 through 35, creates a private measure available for immediate use. Users who connect to the relational database using a configuration with a different login will not be able to access this measure. In all other respects, a user-defined Measure object is identical to a Measure object instantiated in the usual way; it has all the properties and methods of the object class.

The **AddUserMeasure** method requires two arguments, the name of the user-defined measure and its definition. The measure is named **Sales by Half**, a name that is subsequently referred to when instantiating a QueryItem on line 44. The definition itself consists of a simple calculation in which we halve values of the **Units Sold** measure. The syntax for the definition of measures, which can be verified using the **Measure** object's verify property, is documented in the *MetaCube Warehouse Manager's Guide*.

Rather than saving an entire set of Metadata, you can register the measure as available by deploying the **SaveUserMeasure** method, as shown on line 36. The message boxes displayed by lines 37 and 38 both return true values, the first indicating that the new measure is a calculated measure, the second indicating that this measure is user-defined.

Once the measure has been saved, you can immediately define a query incorporating that measure. The QueryItem object on line 43 refers to the user-defined measure in the standard way, by name, and the remainder of the procedure executes the query using syntax that should be familiar from the tutorial at the beginning of this documentation.

The Samples Class of Objects

Sampling improves query performance, extrapolating results from tables that are a fraction of the size of the original fact table. The **Samples** object class describes such tables, which contain a statistically significant, evenly distributed set of records replicated from the fact table.

The Samples Collection's Add Method

After creating a sample table, register that table in MetaCube's metadata by instantiating an object of the **Samples** class. The **Add** method for the Samples collection requires three arguments: the name of the logical object, the name of the table-owner or schema to which the underlying sample table belongs, and the name of the sample table itself.

```
MyMetabase.FactTables.Item(0).Samples.Add _
    "Glenn's Sample", "METADEMO", "SALES_SAMP1"
```

Instantiating a sample object assigns values to several properties of that object, documented below. To remove a Sample object, identify the Samples collection and deploy the **Remove** method, general to all collections, specifying the Sample object to be removed by index number.

Samples Properties

Unlike aggregate tables, the physical structure of which depends on the level of summarization, sample tables always feature the same columns as the fact table, differing only in the number of rows they store. Because the structure of the sample table is not subject to variation, the object that defines the metadata for a sample table has few properties.

Figure 6-9 summarizes the properties of the **Samples** object class.

Figure 6-9
Samples Class of Objects: Properties

Property	Description/Example
DefaultCost	Long integer. Read-only. The default cost of the corresponding table. `c=MySample.DefaultCost`
DefaultRow-Count	Long integer. Read-only. The default row count of the corresponding table. `r=MySample.DefaultRowCount`
Name	String. The name of the Sample object. Like the same property of the Aggregate object, the name of the logical object is largely irrelevant since users and even application developers never need specify a sample by name when defining a query. `MySample.Name = "Hardly Matters"`
Schema	String. The name of the table owner or schema to which the sample table belongs. `MySample.Schema = "METADEMO"`
Table	String. Identifies the underlying sample table by name. Aside from the table's cost, no other information regarding column names or contents need be included for the sample table, since its structure is completely derived from the fact table. `MySample.Table = "SALES_SAMP1"`
TableOptions	String. Includes any SQL syntax to be appended to the CREATE statement executed by MetaCube Sampler when creating the Sample table. `MySample.TableOptions = "EXTENT SIZE 20"`
TableSize	Long. The number of rows in a table. This value enables MetaCube to identify which sample can deliver the specified degree of accuracy for a query. MetaCube does not evaluate the costs of aggregates or of the fact table when processing a query against a sample table. `MySample.TableSize = 4003`
Valid	Boolean. A true value indicates the availability of a sample table. Defaults to `False`. `MySample.Valid = True`

Property	Description/Example
ValueList	ValueList. Read-only. A list of values for this sample, retrieved by MetaCube and stored in the metadata directly; allows applications to display filter choices rapidly. `MsgBox MySample.ValueList.TabbedValues`
Verified	This property stores a long value returned by the **Verify** method indicating that the metadata definition for a Sample object is either: ■ Completely valid ■ Invalid because the Sample object itself is invalid If you have not invoked the **Verify** method, this property defaults to **VerifiedNever**, indicating that the metadata is unverified. The significance of each of the numeric codes stored by the **Verified** property is explained in Figure 3-6 on page 3-16. `MsgBox Sample.Verified`
VerifyResults	This property stores the ValueList returned by the **Verify** method describing any errors in the object's metadata. `MsgBox MySample.VerifyResults.TabbedValues`

The **Samples** object class features no methods and owns no collections. Related properties and methods of **Queries** and **MetaCubes** object classes determine which table MetaCube attempts to extrapolate a result from, as well as the range of error associated with such an extrapolation.

In choosing a sample table, MetaCube weighs both of the following:

- The accuracy specified for the query, represented by the **Accuracy** property of the **Queries** object class
- The table size of the sample relative to other sample tables, represented by the **TableSize** property of the **Samples** object class

The **Accuracy** property of a query stores an integral number between 1 and 100, indicating the relative size of the sample against which MetaCube processes a query. Any value less than the default of `100` prompts MetaCube to process the query against a sample table.

The accuracy requested for the query determines the sample table from which MetaCube should extrapolate a result, but the accuracy value itself bears no absolute statistical relevance. High accuracy values prompt MetaCube to choose the largest available sample tables, low accuracy values prompt MetaCube to choose the smallest available sample tables. If the largest sample table still does not contain a significant fraction of the total records, the margin of error is great even for a query requesting a high accuracy.

The correspondence between accuracy and the sample table chosen depends on the total number of tables and on the rank of the chosen table's size as compared to other sample tables. If there exist five tables sampling the fact table, MetaCube processes queries requesting an accuracy greater than 80 against the largest sample table, queries requesting an accuracy greater than 60 against the second-largest sample table, queries requesting an accuracy greater than 40 against the third-largest sample table, and so on. In this hypothetical example, removing any one of the five sample tables changes the accuracy offered by each of the four remaining tables to a range of 25 rather than 20. Queries requesting a low accuracy process more quickly, as MetaCube can extrapolate results from a smaller table.

A query's assigned accuracy thus directs how MetaCube processes a query, but does not reflect the precision of the extrapolated result. For each result, MetaCube also returns the margin of error. In assigning an error range for each value in the result set, MetaCube evaluates:

- the confidence with which it must predict that the actual result lies within the calculated range, represented by the **Confidence** property of the **Queries** object class.
- the size of the chosen sample as compared to the fact table, represented by the **TableSize** property of the **Samples** and also **FactTables** object classes.
- the relative data density for each attribute included as a QueryCategory: that is, the number of values at the lowest level of the dimensional hierarchy as compared to the level of the specified attribute.

The **CellError** property and the **ErrorVBArray**, **ErrorSpreadClip**, and **FetchCellError** methods of the **MetaCubes** object class retrieve for each value presented in the report the associated range of error. Sampling thus involves three object classes: the **Samples** object class, which defines the metadata for the sample table, the **Queries** object class, which defines queries against sample tables, and the **MetaCubes** object class, which retrieves extrapolated query results and the associated margin of error.

Exercise 18 incorporates all three object classes into a procedure that first defines the metadata for a sample and subsequently extrapolates results from that sample for a query requesting less than complete accuracy.

Although this exercise registers a new sample in MetaCube's metadata, this change is temporary, since the metadata is not saved and the Sample object is abandoned at the close of the procedure. The code for registering a sample in MetaCube's metadata is included here only for completeness; most systems, including the demonstration database, already feature registered sample tables, and queries for which estimates are acceptable are automatically routed to the appropriate sample table.

For this particular procedure to work, a sample table named **SALES_SAMP1** must exist, and the value of the **TableSize** property must be corrected to reflect the number of rows in that table.

Exercise 18: Sampling

```
Option Explicit

'MetaCube API Exercise 18: Sampling


 Sub MetaCube_API()

'Declare Variables
    Dim MyMetabase As Object, MySample As Object, _
    MyQuery As Object, MyMetaCube As Object, _
    MyData As Variant

'Excel Variables
    Dim ReportRange As Range

'Connect
    Set MyMetabase = CreateObject("Metabase")

    'Identify an ODBC Data Source
    Let MyMetabase.ConnectString = "Metademo"

    'Specify a set of metadata
    Let MyMetabase.Name = "MetaCube Demo"

    'Specify login to database
    Let MyMetabase.Login = "metapub"
    'Passes value to database on connection

    'Specify database password
    Let MyMetabase.Password = "Metapub"

    MyMetabase.Connect

'Define Metadata for Sample Table
    Set MySample = _
  MyMetabase.FactTables.Item("Sales Transactions") _
  .SAMPLES.Add("Sample", "METADEMO", "SALES_SAMP1")
    Let MySample.TableSize = 100
    Let MySample.Valid = True
    MyMetabase.Save

'Define Query
    Set MyQuery = MyMetabase.Queries.Add("My New Query")
    MyQuery.QueryCategories.Add "Brand"
    MyQuery.QueryItems.Add "Units Sold"
    Let MyQuery.Confidence = 50
    Let MyQuery.Accuracy = 5
```

```
48      MsgBox MyQuery.SQL 'See query hit sample table
49
50  'Prepare Worksheet
51      Worksheets.Item("Query Report").Activate
52      Cells.Select
53      Selection.ClearContents
54
55  'Get Extrapolated Results
56      Set MyMetaCube = MyQuery.MetaCubes.Add("Data")
57      Let MyData = MyMetaCube.ToVBArray
58      Set ReportRange = ActiveSheet.Range _
59          (ActiveSheet.Cells(1, 1), ActiveSheet.Cells _
60          (MyMetaCube.Rows, MyMetaCube.Columns))
61      Let ReportRange.Value = MyData
62      ReportRange.EntireColumn.AutoFit  'Sizes columns
63
64  'Display Error Just Below Results, Recycle Same Variables
65      Let MyData = MyMetaCube.ErrorVBArray
66      Set ReportRange = _
67          ActiveSheet.Range(ActiveSheet.Cells(1, 3), _
68          ActiveSheet.Cells(MyMetaCube.Rows, _
69          MyMetaCube.Columns + 2))
70      Let ReportRange.Value = MyData
71      ReportRange.EntireColumn.AutoFit  'Sizes columns
72
73  End Sub
```

Explanation of Exercise 18

This procedure begins by declaring the standard set of object variables for connecting to a decision support system, building a query, and defining a report. An additional object variable, **MySample**, is declared on line 9 to store an object of the **Samples** class.

Lines 35 to 37 instantiate a Sample object, assigning the object itself a name, **Sample**, and identifying the table that the object represents, **SALES_SAMP1**, as well as the table-owner or schema to which that table belongs.

Line 38 assigns a value to the **TableSize** property, indicating the number of rows in the table. In the demonstration database, the table **SALES_SAMP1** has 100 rows. Line 39 validates the sample, and line 40 saves the metadata, registering this description of the sample table in the relational database.

Lines 42 to 45 define a query requesting unit sales by brand, a confidence of 50 percent, and an accuracy of 5. Setting a low accuracy ensures that the relatively small sample table be used to process the query. We can confirm that the query accesses data in our new sample table by reviewing the SQL displayed in the message box created by line 48.

The query executes on line 57, which requests that the data be returned from the database as an array. The query result is stored in variant, **MyData**, which populates a range of cells in lines 57 to 60.

The same variables for storing the data and defining a range are given new values in the subsequent section, which begins on line 64. This section of code retrieves the margin of error associated with each value in the query result, displaying the error in a set of cells offset by two columns from the original report. Line 65 converts into an array the error calculated for each extrapolated value, returning zeros for non-numeric cells within the report. Lines 67 to 69 define a range in which to display the error, incrementing the starting and ending positions of the report by 2, a seemingly arbitrary number chosen because the original report includes only two columns and you now want to add another two columns for the error.

Adding the numbers returned by the **ErrorVBArray** method to the values extrapolated for brand sales defines the upper range of MetaCube's estimate; subtracting the same numbers returns the lower range of MetaCube's estimate. As MetaCube based the calculations of these ranges in part on the **Confidence** property of the **Query** object—which in this procedure stores a value of 50—there is at least a 50 percent chance that the precise result lies within this range. In fact, only one of the eight actual brand values falls outside the estimated range.

The SampleQualifiers Class of Objects

Sample objects can own a single collection of objects, those belonging to the **SampleQualifiers** class. Each object of this class stores an index or grant that the sampling agent executes as separate SQL statements following the creation of the sample table. Table options appended to the end of the CREATE statement are stored by the **TableOption** property of the Sample object in MetaCube's programming interface.

To instantiate a SampleQualifier object, specify the SQL statement building an index or granting access privileges and indicate whether that statement should be displayed as an index or a grant in different tools' interfaces:

```
MySample.Qualifiers.Add _
    ("GRANT SELECT ON SAMPLE_TABLE TO USER", "GRANT")
```

The first argument can be any SQL statement stored as a string. The second argument is a string that can store one of two values: `GRANT` or `INDEX`. While MetaCube may accept other string values for the second argument, such values are not registered correctly with Warehouse Manager and Agent Administrator, the applications for managing sample tables.

SampleQualifiers Properties

Objects of the **SampleQualifiers** class have two properties, the values of which are assigned upon instantiation. Figure 6-10 summarizes these properties.

Figure 6-10
SampleQualifiers Properties

Property	Description/Example

Statement	This read-write property stores as a string the SQL statement to be executed following the creation of a sample table.
	`Let MySampleQualifier.Statement = _`
	`"GRANT SELECT ON SAMPLE_TABLE TO USER"`

Tag	This read-write property stores as a string the type of SQL statement represented by the **Statement** property of the SampleQualifier object: `Let MySampleQualifier.Tag = "GRANT"` MetaCube tools understand only two strings, `INDEX` and `GRANT`.

Chapter

7

The Folders Class of Objects

In This Chapter

This chapter introduces the **Folders** class of objects, which provides an interface for storing query and filter definitions. Like UNIX, DOS, or Windows platforms, all of which support hierarchically organized folders or directories, MetaCube folders can be associated with query and filter objects as well as with other folder objects. A folder can own query and filter definitions, but a folder can also own other folders.

The Folders Class of Objects

Like all objects, the Folder object simply provides an interface for MetaCube functionality. The actual definitions of queries and filters continue to be stored in MetaCube's metadata tables, but the folder programming interface offers developers and users a familiar paradigm for organizing that information.

Although the idea of folders might be completely familiar to users of most operating systems, as an object in MetaCube's OLE library they differ in three respects from other object classes:

- The **RootFolder** object owns the first collection of **Folder** objects. A single **RootFolder** object belongs to each instance of a **Metabase** object. You cannot instantiate another **RootFolder** object.
- The only collection belonging to a **Folder** object is another collection of **Folder** objects. Because each **Folder** object can, in turn, own a collection of **Folder** objects, the number of collections is unlimited.
- The level of a collection of **Folder** objects varies, with some collections belonging directly to the **RootFolder** object, and others removed from the **RootFolder** object by one or more collections.

As shown in the next section, the varying parentage of the **Folders** class of objects complicates the syntax for instantiating an object of this class.

Instantiating a Folder Object

As noted previously, each Metabase object's root folder owns a collection of folders, and each folder can in turn can own a collection of folders, and so on. The syntax for instantiating a Folder object depends on the parent of the Folder object:

```
MyMetabase.RootFolder.Folders.Add "My Objects"
MyMetabase.RootFolder.Folders.Item _
    ("My Objects").Folders.Add "New Filters"
MyMetabase.RootFolder.Folders.Item _
    ("My Objects").Folders.Item _
("New Filters").Folders.Add "Time Filters"
```

Storing instantiations of Folder objects in object variables simplifies the syntax for creating subfolders or subdirectories, achieving an identical result with better performance:

```
Set Level1Folder = MyMetabase.RootFolder.Folders.Add _
    ("My Objects")
Set Level2Folder = Level1Folder.Folders.Add ("New Filters")
Set Level3Folder = Level2Folder.Folders.Add ("Time Filters")
```

Once you have instantiated a folder, users can house saved queries in that folder, specifying the Folder object itself as an argument to save methods for queries and filters, as documented in Figure 3-2 on page 3-12 and "Filters Methods" on page 8-37.

To eliminate a folder, deploy the collection's **Remove** method, identifying the folder to be removed by name or by index number. To identify the names of the folders in a particular collection, retrieve the contents of the **Names** property of the Folders collection, a ValueList that defaults to a tab-delimited string. As a collection, Folder objects possess the same properties and methods as other object collections, which are described in "Object Class Hierarchies and Collections" on page 1-6.

Folder Properties

The properties of a Folder object store the name of the folder and the names of the queries and filters housed in that folder. Figure 7-1 summarizes the properties of the **Folders** class of objects.

Figure 7-1
Folders Class of Objects: Properties

Property	Description/Example
FilterNames	This read-only property stores the names of filters associated with a folder, an owner, and a group. The owner is the login of the user who saved those filters. The arguments are the name of the owner and the name of the group, both as strings. To view the values of this property, enter a pair of empty double quotes for the group argument. A ValueList is returned, which defaults to a tab-delimited string. `MsgBox MyFolder.FilterNames "MetaDemo", "Time"`
Name	Stores the name of the folder. `Let MyFolder.Name = "Glenn"`
QueryNames	This read-only property stores the names of queries associated with a folder and an owner. The owner is often the login of the user who saved the queries. You must specify the name of the owner as a string argument to this property. Returns a ValueList, which defaults to a tab-delimited string. `MsgBox MyFolder.QueryNames "MetaDemo"`

Folders Methods

The **Folders** class of objects features three methods, one for providing the path to a Folder object, one for renaming queries associated with a Folder object, and the other for renaming filters associated with a Folder object. Figure 7-2 summarizes the methods of the **Folders** class of objects.

Figure 7-2
Folders Class of Objects: Methods

Property	Description/Example
FullPathName	This method returns the full path to a Folder object, which may be useful when manipulating mandatory filters (which are typically assigned in MetaCube Secure Warehouse). You must specify as an argument the name of the Folder object for which a full path is needed. `UserFolderPath = MyFolder.FullPathName("AnotherFolder")`
RenameFilter	Substitutes a new name, group, and/or owner for any one of the Filter objects housed in that folder. The method requires six arguments: the first three are the current owner, group, and name of the filter, the second three are the new owner, group, and name of the filter. `MyFolder.RenameFilter _ "informix", "Product", "Brand Filter",_ "metapub", "Time", "Week Filter"`
RenameQuery	Substitutes a new name and/or a new owner for any one of the Query objects housed in that filter. The method requires four arguments: the first two are the current owner and name of the query, the second two are the new owner and name of the query. `MyQuery.RenameQuery "informix", "Old Query", _ "metapub", "New Query"`

Exercise 19 provides an overview of the major object classes, methods, and properties necessary for creating folders, saving queries and filters in those folders, renaming filters and queries, and, finally, opening those objects again.

Exercise 19: Saving Queries and Filters to Folders; Renaming, Opening Queries and Filters from Folders

```
1  Option Explicit
2
3  'MetaCube API Exercise 19: Saving Queries and Filters to
   Folders;
4  '    Renaming, Opening Queries and Filters from Folders
5
6  Sub MetaCube_API()
7
8  'Declare Variables
9  Dim MyMetabase As Object, Level1Folder As Object, _
10     Level2Folder As Object, Level3Folder As Object, _
11     MyQuery As Object, MyFilter As Object, _
12     SavedQuery As Object
13
14 'Login as MetaDemo
15 Set MyMetabase = CreateObject("Metabase")
16
17 'Identify an ODBC Data Source
18 Let MyMetabase.ConnectString = "Metademo"
19
20 'Specify a set of metadata
21 Let MyMetabase.Name = "MetaCube Demo"
22
23 'Specify login to database
24 Let MyMetabase.Login = "metademo"
25 'Passes value to database on connection
26
27 'Specify database password
28 Let MyMetabase.Password = "MetaDemo"
29
30 MyMetabase.Connect
31
32 'Create Folders
33 Set Level1Folder = MyMetabase.RootFolder.Folders.Add _
34     ("Objects Folder")
35 Set Level2Folder = Level1Folder.Folders.Add _
36     ("Filter Objects Folder")
37 Set Level3Folder = Level2Folder.Folders.Add _
38     ("Product Filter Folder")
39
40 'Define and Save Query
41 Set MyQuery = MyMetabase.Queries.Add("New Query")
42 MyQuery.QueryCategories.Add "Brand"
43 MyQuery.QueryItems.Add "Units Sold"
44 MyQuery.SaveAs "API Query", Level1Folder
```

```
45
46  'Define and Save Filter
47  Set MyFilter = MyQuery.Filters.AddNewFilter _
48                  ("Alden Filter")
49  Let MyFilter.Group = "Product"
50  MyFilter.FilterElements.Add _
51                  "Brand", "=", "'Alden'"
52  MyFilter.SaveAs "Alden Filter", Level3Folder
53
54  'See List of Queries and Filters Saved in Different Folder
55  MsgBox MyMetabase.RootFolder.Folders.Names
56  MsgBox Level1Folder.QueryNames("metademo")
57  MsgBox Level3Folder.FilterNames("metademo", "")
58
59  'Rename Queries and Filters
60  Level1Folder.RenameQuery _
61              "metademo", "API Query", _
62              "metapub", "Public Query"
63  Level3Folder.RenameFilter _
64              "metademo", "Product", "Alden Filter", _
65              "metademo", "Product", "Alden Filter1"
66
67  'Display New Contents of Folders:  Objects by metapub
68  MsgBox Level1Folder.QueryNames("metapub")
69  MsgBox Level3Folder.FilterNames("metademo", "")
70
71  End Sub
```

Explanation of Exercise 19

The exercise begins by declaring object variables to store Metabase, Query, Filter, and Folder objects. Two object variables, **MyQuery** and **SavedQuery**, are declared to store Query objects: one defines and saves the original instance of the **Queries** class of objects, the second stores the definition of that same query as retrieved from the metadata. Since the original query definition remains in memory, the second object variable is unnecessary. Two different object variables are used for clarity, demonstrating explicitly how to save and open queries in a single procedure.

Lines 14 to 30 instantiate a Metabase object and establish a connection between MetaCube and the demonstration database. The **Login** property of the Metabase object explicitly identifies the user that will later be used as the default author of all saved queries and filters.

After connection, the procedure creates several folders for storing saved queries and filters. Each Metabase object owns a RootFolder object, which in turn owns a collection of Folder objects. Line 33 instantiates a Folder object in this collection called **Objects Folder**. In turn, line 35 instantiates a Folder object owned by the **Objects Folder** called **Filter Objects Folder**. Line 37 instantiates a third Folder object as a subdirectory of that **Filter Objects Folder**. For convenience, each Folder object is assigned to an object variable. When saving and opening queries and filters, the objects stored in these object variables are referred to as arguments to different methods.

Lines 41 to 43 define a simple query, and line 44 saves the query under the name **API Query** in the folder titled **Objects Folder**. Similarly, lines 47 to 51 define a filter on brands, assigning the filter to a group including other filters on attributes of the product dimension. Line 52 saves the filter in the folder titled **Product Filter Folder**. Since you saved the query before defining the filter, the saved query definition does not include any filters. When opening the query again, you have to open the filter as well as apply that filter to the opened query.

Line 55 displays in a message box the names of the folders in the collection owned directly by the RootFolder object. Unless folders have been created previously, only one folder, Objects Folder, belongs to that collection. Line 56 displays the queries within this folder that were saved to the database by the **metademo** user. Line 57 identifies the filters saved in this folder by the **metademo** user. Passing an empty string as the second argument to this property includes filters from all groups, regardless of the dimension to which they belong. All arguments are enclosed in parentheses for these properties because each property returns values to a message box procedure.

Lines 60 to 62 rename the query saved on line 44, first specifying its original owner and name and then assigning a new owner, **metapub**, and a new name, **Public Query**, to the query definition. Switching the owner to **metapub** makes the query available to all Explorer and MetaCube for Excel users. Similarly, lines 63 to 65 rename the filter saved on line 52, assigning a different owner to the filter but otherwise leaving the name and the group unchanged.

Lines 68 and 69 confirm that the author of both query and filter have changed, polling the metadata to determine the names of any queries and filters saved by the **metapub** user to **Objects Folder** and **Product Filter Folder**, respectively.

Chapter

8

The Queries Class of Objects and Related Collections

In This Chapter

This chapter explains the **Queries** and **QueryBackJobs** classes of objects and their related properties, methods, and collections. The sample exercises included in the tutorial at the beginning of this reference demonstrate how you can deploy many of these objects in your own applications.

The Queries Class of Objects

Once you have described the tables in your data warehouse as multidimensional MetaCube objects, you can define queries to retrieve information from those tables. A query definition refers to the measures, dimension elements, attributes, and filters defined by previous objects of the same name.

Query definitions that include a measure retrieve transaction-based data; queries that include only attributes or dimension elements from a given dimension retrieve data about the relationships within the dimensional hierarchy. Transaction-based queries must include at least one attribute or dimension element by which to group transactions. Such queries can feature an unlimited number of attributes or dimension elements from an unlimited number of dimensions.

Figure 1-4 on page 1-13 summarizes each component of a query's definition. Each query belongs to a collection descended from a particular instantiation of a Metabase object and is defined in terms of the multidimensional view inscribed by a DSS System.

Instantiating a Query Object

To instantiate a Query object, simply add a new instance of the Queries class of objects to a Metabase object's collection of queries:

```
MyMetabase.Queries.Add "A Brand-New Query"
```

When instantiating a query you must include the name of the query as an argument. When Explorer instantiates a new query, the application generates a generic, sequentially numbered name, such as **Untitled1**.

Queries Properties

The Queries class of objects actually represents a collection of attributes, measures, filters, and multidimensional results. As such, this master object's properties and methods allow you to generally characterize and manipulate collections of attributes, measures, filters, and results as a single entity that defines the data you want to retrieve from a database and how you want to summarize, pivot, and present that data.

The data a Query object retrieves is thus defined by the objects within its various collections; its properties reflect characteristics, such as the performance cost of processing the query, the name of the query and its author, the availability of the query's result, and its definition in the metadata tables. Figure 8-1 summarizes the properties of the Queries class of objects.

Figure 8-1
Queries Class of Objects: Properties

Properties	Description/Example
Accuracy	Long. A number between 1 and 100 indicating the degree of accuracy with which MetaCube should attempt to process a query. Any value less than 100 prompts MetaCube to process the query against a sample table. The range of accuracy requested for the query directs MetaCube to extrapolate a result from a larger or smaller sample table, as available, but the accuracy value itself bears no absolute statistical relevance. Defaults to 100. `MyQuery.Accuracy = 60` For more information about sampling, accuracy, and margins of error, as well as a complete example, see "The Samples Class of Objects" on page 6-30.
Author	String. Identifies the author of a query; defaults to the name of the database user provided at login. `MyQuery.Author = "informix"`
Confidence	Long. A number between 1 and 100 determining the statistical confidence with which MetaCube will report results extrapolated from sample tables. As the value of this property increases, the range of error values also increases. For example, MetaCube might be able to predict with 80 percent confidence that a number falls between 210 and 230, but it can predict with only 50 percent confidence that the number falls between 215 and 225. Defaults to 100. The **CellError** property and the **FetchCellError**, **ErrorVBArray**, and **ErrorSpreadClip** methods of the **MetaCubes** object class return the margin error for a sampled report. See "The Samples Class of Objects" on page 6-30 for a more detailed explanation of sampling and extrapolated query results. `MyQuery.Confidence = 50`

(1 of 4)

Properties	Description/Example

Cost	Long Integer. The relative performance cost of executing this query is returned by MetaCube's optimizer for any valid transactional query before or after execution. This value reflects the size of the aggregate table, fact table, or sample table chosen by the MetaCube engine to answer the query. Although this value generally corresponds to the number of rows in the table, its scale ultimately depends on how the metadata for those tables has been defined. Costs are additive; the cost of queries consisting of multiple SQL statements or SQL statements accessing multiple fact or aggregate tables is the sum of the costs of the tables accessed. Read-only.
	`If MyQuery.Cost > 1000 _`
	`  Then MsgBox "Query may involve delay."`

CurrentData	Boolean. A true value indicates that MetaCube has executed the query as currently defined and the result remains resident in local memory; a false value indicates otherwise. Read-only.
	`If MyQuery.CurrentData = True _`
	`  Then MsgBox "Query result in memory."`

DataSource	String. Stores the name of the data source for which a query is currently being defined. This property only stores information to which an application may refer after opening a saved query, thereby determining which dimensions and measures to display. MetaCube neither assigns a default value to this property nor uses this information in any way when generating SQL for a query. Properly scoped, each QueryCategory and QueryItem identify the data source from which an attribute or measure is drawn; see "Scoping Rules" on page 14-3. Similarly, you can determine the dimensions, dimension elements, and attributes available for each data source by referring to the collection of Dimension objects owned by a particular FactTable object, as documented in "The Dimensions Class of Objects, as Owned by a FactTable Object" on page 6-22. `Let MyQuery.DataSource = "Sales Transactions"`
ExecutionTime	Date variant. Indicates the length of time required to execute the query, incremented from 12:00:00 A.M. Null pending execution. Read-only. `MsgBox MyQuery.ExecutionTime`

(2 of 4)

Properties	Description/Example
Folder	Object. Read-only. Represents the Folder object under which the Query object has been saved. This property is invalid for Query objects that have not been saved. `MsgBox MyQuery.Folder.Name`
Item-Orientation	Long. Determines whether the data returned by the Query object is organized by rows or columns. Figure 8-6 on page 8-24 lists the appropriate constants. `MyQuery.ItemOrientation = OrientationColumn`
LastUpdate	Date variant. Indicates date and time of query's most recent change. Read-only. `MsgBox MyQuery.LastUpdate`
Live	Boolean. Indicates whether a connection has been established to execute the query as currently defined. Defaults to True, requiring a Metabase connection. This property stores a False value in cases in which the query has been defined off-line. Read-only. `MsgBox MyQuery.Live`

Maximum-Exceeded	Boolean. Indicates whether the query, upon execution, returned a number of rows exceeding the maximum allowed by the Metabase object's **MaxTotalFetches** property. Defaults to false.
	`If MyQuery.MaximumExceeded = True _`
	`Then MsgBox "Too many rows..."`

Name	String. The query object's name, which is specified as an argument upon instantiation. Default property. `MyQuery.Name = "Untitled1"`
Parameters	ValueList. Stores as a read-only list all filter parameters undefined for the query. Parameters previously defined by Agent Administrator, the application, or the user are not included in this list. To review an application that deploys parameters, see on page 8-14. `MsgBox = MyQuery.Parameters`
Parent	Object. The Metabase object. `MsgBox MyQuery.Parent.Name`

(3 of 4)

Properties	Description/Example
ResultSource	Long. This read-only property indicates the source used for the results of the query. ■ 0: Unknown. This is the default. ■ 1: Sampling used to retrieve the results of the query. ■ 2: Aggregate table used to retrieve results of the query. ■ 3: Fact table used to retrieve the results of the query; neither sampling nor aggregate table are used. `MyQuery.ResultSource=0`
Saved	Boolean. `True` indicates that the query as currently defined is saved in the database; `False` indicates otherwise. `If MyQuery.Saved = False Then MyQuery.Saved`
SQL	ValueList. Stores the SQL generated by the MetaCube analysis engine for any valid query, with each SQL statement representing a separate string in an array of values. For each entity included in the COMPARE expression of a QueryCategory and for each different data source included in a query, MetaCube generates a separate SQL statement, as described in "Compare" on page 5-44. `MsgBox MyQuery.SQL`
TimeFiltered	Boolean. True indicates that the query includes a filter on time; false indicates otherwise. Read-only. `MsgBox MyQuery.TimeFiltered`

(4 of 4)

Queries Methods

Figure 8-2 summarizes Queries object class methods.

Figure 8-2
Queries Class of Objects: Methods

Method	Description/Example
AsynchLast-Error	Returns a string containing the exception message generated when an asynchronous query fails. An asynchronous query is generated by the **AsynchRetrieve** method. `MsgBox MyQuery.AsynchLastError`
AsynchRetrieve	Explicitly commands MetaCube to transmit the SQL generated for a query directly to the relational database and to do so in a separate thread so the query executes asynchronously, freeing the user from waiting for the query's results. Except for initiating asynchronous execution, this method otherwise behaves exactly like **Query.Retrieve**. If a user is limited to mandatory QueryBack jobs, the same limitations apply when that user attempts to submit an asynchronous query. **Note:** Only one asynchronous query per Query object can run at any given time. While an asynchronous query is running, no method can change the definition or execution of a Query object. `MyQuery.AsynchRetrieve`
Asynch-RetrieveStatus	Returns a string showing the status of an asynchronous query (a query transmitted to the relational database using **Query.AsynchRetrieve**). If the asynchronous query is retrieving rows, this method returns a row count. If there is no asynchronous query outstanding or an asynchronous query fails, the returned string is empty. To retrieve a string containing the last exception message generated when an asynchronous query fails, use the **AsynchLastError** method. `MsgBox MyQuery.AsynchRetrieveStatus`
CancelAsynch-Retrieve	Cancels an asynchronous query. Because only one asynchronous query can execute at any given time, no arguments are necessary for this method. This method does not complete until the asynchronous query has stopped executing. `MyQuery.CancelAsynchRetrieve`

(1 of 5)

Method	Description/Example
Clear-Parameters	Erases any values a procedure previously assigned to the parameters included in a query. This method does not enable the application to bypass or erase parameter values created in Agent Administrator. Those values always override parameters assigned by the application. `MyQuery.ClearParameters`
GetParameter-Object	Returns the QueryCategory included in the filter definition for a specified parameter. Since the QueryCategory object's default property, **Category**, stores a self-descriptive string, this method is useful for identifying the attribute for which values must be supplied to complete the filter. To review an application that invokes this method, see "Exercise 20: Executing Queries That Include Parameterized Filters" on page 8-14. The method requires one argument, an index number identifying the parameter for which the QueryCategory object is sought. `MyQuery.GetParameterObject 1`
GetParameter-Operator	Returns a string containing the operator for an undefined parameter. Since more than one parameter may be undefined for a given query, you must identify the undefined parameter by index number. This method is useful for precluding users from submitting multiple values for a parameter using an "=" operator. `MsgBox MyQuery.GetParameterOperator (0)`
OpenStorage	Returns a Query object, along with all children and any data stored as a query result, from an IStorage object. An IStorage object can store information in almost any location, including a local file. This method is available only in C++ and other development environments that directly support the COM interface. The method accepts one argument, an IStorage object, which you can initialize according to your custom storage needs. Once the object has been initialized, the query can be saved in an IStorage object using the **SaveStorage** method of the **Queries** class of objects.

(2 of 5)

Method	Description/Example
Retrieve	Explicitly commands MetaCube to transmit the SQL generated for the query directly to the relational database. Before execution, the query must either consist of one QueryItem and one QueryCategory, or one or more QueryCategory objects specifying an attribute from the same dimension. Several methods of the **MetaCube** class of objects, including **ToVBArray** and **ToSpreadClip**, can implicitly require query execution, rendering deployment of this method unnecessary. `MyQuery.Retrieve`
Save	Saves the query's definition in metadata tables on the relational database under the name and author specified by the Query object's corresponding properties. Consequently, no arguments are required. `MyQuery.Save`
SaveAs	Saves the query's definition in metadata tables on the relational database under a different name than that specified by the Query object's **Name** property. This method requires you to specify the query's name as a string and the folder under which the query definition will be saved as an object. Two queries with the same name cannot be saved to a single folder. For a complete example of saving and opening queries and filters into and from different folders see "Exercise 19: Saving Queries and Filters to Folders; Renaming, Opening Queries and Filters from Folders" on page 7-7. `MyQuery.SaveAs "Sales Report", MyMetabase.RootFolder`
SaveStorage	Saves a Query object as an IStorage object, including all the children of that query as well as any data associated with the query. An IStorage object can store information in almost any location, including a local file. This method is available only in C++ and other development environments that directly support the COM interface. The method accepts one argument, an IStorage object, which you can initialize according to your custom storage needs.

(3 of 5)

Method	Description/Example
SetParameter	Assigns a value or values to a filter parameter. The method requires two arguments: the index number of the parameter, which depends on the order in which the FilterElement object has been instantiated, and the values to be substituted for that parameter, listed in a single string. The string should conform to SQL standards, listing values in parentheses, with each value in single quotes separated by a comma, as shown. `MyQuery.SetParameter 0, "('Alden', 'Delmore')"` For a complete example, see "Exercise 20: Executing Queries That Include Parameterized Filters" on page 8-14. When submitting parameterized queries to QueryBack, undefined parameters must be stored in the metadata for future reference by the UNIX process **clientexec**. To identify and define parameters in MetaCube's metadata, launch Agent Administrator. Parameters defined through Agent Administrator and registered in MetaCube's metadata take precedence over any parameter assignments made by the **SetParameter** method.
SetSQL	Replaces an SQL statement generated by MetaCube's analytical engine with a string specified as an argument. Because **COMPARE** expressions or multiple fact table queries might result in multiple SQL statements for a given query, you must identify the SQL statement to be replaced by index number, with the first statement identified by a zero. The method thus requires two arguments, the index number, an integer, and the replacement SQL statement, a string. `MyQuery.SetSQL 0, "SELECT…"`

Method	Description/Example

Submit	Submits the query to MetaCube's QueryBack agent for background processing, instantiating a QueryBackJob object. You cannot submit a query for background processing if any of the filters included in that query have not been saved.
	
	The **Submit** method requires four arguments: a string identifying the name of the QueryBack job; a numeric integer indicating the query's priority, as compared to other queries that may be queued for background processing; the time at which you would like the server to process the query; and a numeric argument indicating the frequency with which the query should be re-executed, if ever. To assign a high priority to a query, specify a high number as a priority argument. MetaCube Explorer limits users to a priority of 5, but MetaCube does not. Figure 8-3 on page 8-20, explains the significance of each frequency argument. The arguments for this method are arranged in the format:
	
	`Set MyQueryBackJob = Query.Submit Name (string), _`
	`Priority (integer), TargetStartTime (date variant), _`
	`RecurType (integer)`
	
	For an example of this method see "Exercise 21: Submitting a Query to QueryBack" on page 8-17. For a discussion of how to track and retrieve QueryBackJob results, see "The QueryBackJobs Class of Objects" on page 8-78.

(5 of 5)

The following two exercises describe particularly complex procedures for submitting queries to the database. The first example, "Exercise 20: Executing Queries That Include Parameterized Filters" on page 8-14, demonstrates how to define a query that includes a parameterized filter. The second example, "Exercise 21: Submitting a Query to QueryBack" on page 8-17 demonstrates how to submit a query to QueryBack.

Exercise 20: Executing Queries That Include Parameterized Filters

```
Option Explicit

'MetaCube API Exercise 20: Executing Queries That Include
Parameterized Filters

Sub MetaCube_API()

'Declare Variables
    Dim MyMetabase As Object, MyQuery As Object, _
        MyFilter As Object, MyMetaCube As Object, _
        MyData As Variant, Count As Integer, _
        Parameters As Variant, _
        ParameterValues As Variant

'Excel Variables
    Dim ReportRange As Range

'Connect
    Set MyMetabase = CreateObject("Metabase")

    'Identify an ODBC Data Source
    Let MyMetabase.ConnectString = "Metademo"

    'Specify a set of metadata
    Let MyMetabase.Name = "MetaCube Demo"

    'Specify login to database
    Let MyMetabase.Login = "metademo"
    'Passes value to database on connection

    'Specify database password
    Let MyMetabase.Password = "Metademo"

    MyMetabase.Connect

'Define Query
    Set MyQuery = MyMetabase.Queries.Add("My New Query")
    MyQuery.QueryCategories.Add "Brand"
    MyQuery.QueryItems.Add "Units Sold"

'Define Parameterized Filters
   MyQuery.ClearParameters
    Set MyFilter = MyQuery.Filters.AddNewFilter _
        ("Brand & Region Parameters")

```

```
45      MyFilter.FilterElements.Add "Brand", "=", _
46        "<<Please enter one brand to filter on...>>"
47
48      MyFilter.FilterElements.Add "Region", "=", _
49        "<<Please enter one region to filter on...>>"
50
51      MsgBox MyQuery.Parameters  'All undefined parameters
52      Let Parameters = MyQuery.Parameters.ArrayValues
53       'Stores both parameters as array
54
55      For Count = 0 To UBound(Parameters) 'to # of param.
56       Let ParameterValues = InputBox _
57         (Title:=MyQuery.GetParameterObject(Count), _
58         Prompt:=Parameters(Count))
59       MyQuery.SetParameter Count, "'" + ParameterValues + "'"
60      Next Count  'Go back and do the next parameter
61
62      MsgBox MyQuery.Parameters
63      'All parameters defined, nothing to show
64
65  'Get Results
66      Worksheets.Item("Query Report").Activate
67      Cells.Select
68      Selection.ClearContents
69
70      Set MyMetaCube = MyQuery.MetaCubes.Add("Data")
71      Let MyData = MyMetaCube.ToVBArray
72      Set ReportRange = ActiveSheet.Range _
73          (ActiveSheet.Cells(1, 1), _
74          ActiveSheet.Cells _
75          (MyMetaCube.Rows, MyMetaCube.Columns))
76      Let ReportRange.Value = MyData
77      ReportRange.EntireColumn.AutoFit
78
79  End Sub
```

Explanation of Exercise 20

This procedure defines a filter consisting of two constraints, each represented by a different FilterElement object. The syntax for instantiating those objects includes three arguments: the name of the attribute from which values are being selected; the operator; and the values themselves, also called the operand. In this example, however, parameters appear in place of the operand, enabling the user to specify different attribute values for the filter at runtime. A parameter, which can be any string, appears bracketed in a pair of less-than and greater-than symbols on lines 46 and 49.

The **Parameters** property of the Query object stores a list of undefined parameters, comprised of parameters for which values must be specified prior to execution of the query. Once a value has been substituted for a parameter, that parameter is no longer included in the list. The message box created on line 51 thus includes two parameters, whereas the message box created on line 62 displays none, indicating that attribute values have been substituted for every parameter.

Because the **Parameter** property of the Query object stores the entirety of every parameter label included in the ValueList, we can store those parameters in an array, as represented by the variant variable **Parameters**, subsequently displaying each parameter label as a prompt in a dialog box. Lines 55 to 60 iterate through the parameters in the array, displaying in a dialog box the parameter label as a prompt. The title bar of the dialog box displays the name of the attribute for which parameter values are sought, obtained by the **GetParameterObject** method on line 57.

The **SetParameter** method substitutes the entered value for the parameter. This method requires two arguments:

- The index number of the parameter, in this case identified by the loop counter
- The SQL syntax for a list of attribute values, as a string variable called **ParameterValues**. Acceptable syntax is of the form "('Value1', 'Value2',…)"

When generating SQL for a query, MetaCube retrieves data only for the specified attribute values.

Parameters set by an application remain set only as long as the application maintains a connection to MetaCube. Parameters defined by an application no longer apply to QueryBack jobs, which require all parameters to be saved in the metadata through MetaCube's Agent Administrator application. There is no programmatic interface for defining QueryBack parameters.

The following exercise illustrates how to submit a query to QueryBack. You must create a configuration for connecting to a server-side database running MetaCube Agents to execute this procedure.

Exercise 21: Submitting a Query to QueryBack

```
1  Option Explicit
2
3  'MetaCube API Exercise 21: Submitting a Query to QueryBack
4
5   Sub MetaCube_API()
6
7  'Declare Variables
8  Dim MyMetabase As Object, _
9  MyQuery As Object, _
10 MyMetaCube As Object, _
11 MyQueryBackJob As Object, _
12 QBQuery As Object, _
13 QBCube As Object, _
14 ReportRange As Range, _
15 MyData As Variant
16
17 'Declare Constants
18 Const Priority = 4
19 Const RecurTypeNone = 1
20 Const QueryBackJobStatusPending = 0
21 Const QueryBackJobStatusFinished = 2
22
23 'Connect
24 Set MyMetabase = CreateObject("Metabase")
25
26 'Identify an ODBC Data Source
27 Let MyMetabase.ConnectString = "Metademo"
28
29 'Specify a set of metadata
30 Let MyMetabase.Name = "MetaCube Demo"
31
32 'Specify login to database
33 Let MyMetabase.Login = "metademo"
34 'Passes value to database on connection
35
36 'Specify database password
37 Let MyMetabase.Password = "Metademo"
38
39 MyMetabase.Connect
40
41 'Build Query
42 Set MyQuery = MyMetabase.Queries.Add("New Query")
43 MyQuery.QueryItems.Add "Units Sold"
44 MyQuery.QueryCategories.Add "Brand"
45
46 'Add cube to MyQuery's cube collection
47 Set MyMetaCube = MyQuery.MetaCubes.Add("New Cube")
```

```
Let MyMetaCube.Scratch = False

'Submit to QueryBack
Set MyQueryBackJob = MyQuery.Submit _
(MyQuery.Name, Priority, MyMetabase.CurrentTime, _
                RecurTypeNone)
'Query scheduled to run on the server, as soon as possible

'Wait for Query to Finish
Check:
MyQueryBackJob.refreshstatus
MsgBox Prompt:=MyQueryBackJob.Status, Title:="Job Status"
MsgBox "Check the status of the QueryBack job now?"
MyQueryBackJob.refreshstatus
If MyQueryBackJob.Status <> QueryBackJobStatusFinished Then
       MsgBox "Your Job has not finished. Check later."
       Exit Sub
End If

'Get QueryBackJob, Store Query in QBQuery
Set QBQuery = MyQueryBackJob.Retrieve

'Cube and Query returned by QueryBack
MsgBox QBQuery.MetaCubes.Count
Set QBCube = QBQuery.MetaCubes.Item(0)

'Format data as an array VB can display, store in variable
Let MyData = QBCube.ToVBArray

'Clear "Query Report" Worksheet
Sheets("Query Report").Activate
Cells.Select
Selection.ClearContents

'Excel Code:  Defines Range of Cells, Presents Data
Worksheets.Item("Query Report").Activate
Set ReportRange = _
    ActiveSheet.Range(ActiveSheet.Cells(1, 1), _
    ActiveSheet.Cells(QBCube.Rows, QBCube.Columns))
Let ReportRange.Value = MyData
ReportRange.EntireColumn.AutoFit  'Sizes columns

End Sub
```

Explanation of Exercise 21

This exercise instantiates a Query object as **MyQuery** and defines it by one measure and one attribute, **Units Sold** and **Brand**, respectively.

Lines 42 to 44 define the query, storing an instance of the Query object in the object variable in **MyQuery**. Lines 47 and 48 also instantiate a MetaCube object and set its scratch property to false, enabling MetaCube to store all cubes associated with a query when the query is saved or submitted to QueryBack.

Line 51 submits the job for background query processing, using the **Submit** method of the Query object to instantiate a new QueryBackJob object. To instantiate the QueryBackJob object, you must specify the name of the job, the job's priority, the time at which the job should run, and how often the job should recur.

The name of the QueryBackJob is, in this case, specified using the **Name** property of **MyQuery**, although QueryBackJob names can differ from Query names. The query has a priority of 4 and is set not to recur. The **CurrentTime** property of the Metabase object returns the time from the PC clock, instructing the QueryBack agent to process the query as soon as possible.

Once the query has been submitted, you must periodically review the job queue to determine its status. Lines 57 to 65 define a loop that recurs until the job finishes, using the **RefreshStatus** method of the QueryBackJob object to poll the queue.

Line 68 executes once the status of the QueryBackJob indicates that the job has finished. The **Retrieve** method returns a new Query object, storing the new object in a new variable, **QBQuery**. Line 71 displays a message box confirming that the new Query object returned by the **Retrieve** method also includes a MetaCube object. Line 75 stores this object in the **QBCube** object variable. New object variables are declared for both Query and MetaCube objects to demonstrate that neither object was merely resident in memory and that the **Retrieve** method returned genuinely new Query and MetaCube objects. The rest of the query is displayed in a spreadsheet as before.

For more information about QueryBack jobs, see "The QueryBackJobs Class of Objects" on page 8-78.

Related Constants

Figure 8-3 summarizes the numeric values for the **Submit** method's frequency argument.

Figure 8-3
QueryBack Frequency Constants

Frequency	MetaCons.bas Constant Name	Constant
Once	RecurTypeNone	1
Daily	RecurTypeDaily	2
Weekly	RecurTypeWeekly	3
Monthly	RecurTypeMonthly	4
Annually	RecurTypeAnnually	5

Queries Collections

The Query object's collections define the components of a query, as depicted in Figure 1-2 on page 1-9. Of the four collections, three incorporate references to MetaCube's metadata, with QueryCategory objects identifying the attributes by which a query groups or summarizes transactional data, QueryItem objects identifying the measures a query retrieves, and Filter objects identifying the set of constraints to place on the range of data a query retrieves. MetaCube objects represent ways of organizing, summarizing, and presenting the data a query retrieves.

Figure 8-4 summarizes the collections of a Query object.

Figure 8-4
Queries Class of Objects: Collections

Collections	Description/Example
DrillData-Sources	Consists of the FactTable objects that can be accessed by a query. A query can access information in different fact tables only if the data in those fact tables can be grouped by the attributes currently included in the query. As different QueryCategory objects incorporate attributes into a query definition, the MetaCube analysis engine excludes FactTable objects from the DrillDataSources collection on the basis of the Dimension objects contained in each FactTable's DimensionMappings collection. Applications cannot directly instantiate a new item in this collection, and no add method is available. This collection is thus available only for reference. For an overview of the properties of a given FactTable object within this collection, see Figure 6-1 on page 6-4. `MsgBox MyQuery.DrillDataSources.Item (0).Name`
Filters	Consists of ad hoc or saved filters that have been applied to the query's definition. Any ad hoc filters designed for a particular query exist only within this collection until saved. `MyQuery.Filters.Add "This Week"`
MetaCubes	Consists of different multidimensional representations of a query result, often referred to as virtual cubes of data. MetaCube does not, however, store data in a physical cube. `MyQuery.MetaCubes.Add "Break Report"`
Query-Categories	Consists of objects that identify the names of Attribute and/or DimensionElement objects by which the query groups and summarizes transactional data in the report. `MyQuery.QueryCategories.Add "Brand"`
QueryItems	Consists of objects that specify the type of numeric data the query retrieves; refer to the names of Measure objects. `MyQuery.QueryItems.Add "Units Sold"`

The QueryCategories Class of Objects

A QueryCategory object specifies an attribute or expression to include in a query's definition. Attribute objects, discussed in "The Attributes Class of Objects" on page 4-16, describe physical columns in the relational database storing string values by which transactional data is characterized and grouped. Organized in dimensions, attributes typically describe different levels within a hierarchy of increasingly summarized values. From the collection of available attributes owned by the Metabase object or from the collections of available attributes owned by the Dimension objects associated with a fact table, a QueryCategory identifies an Attribute object to apply to a query. A QueryCategory object can also represent some function performed on an Attribute object, such as a **BUCKET** or a **COMPARE** function.

To include an attribute in a query's definition, simply instantiate a QueryCategory object, identifying by name the Attribute object defined as a component of the Metabase object's metadata:

```
MyQuery.QueryCategories.Add "Brand"
```

The case-sensitive attribute name is stored as a string in the QueryCategory object's only unique property, the **Category** property. Although MetaCube Explorer does not allow users to incorporate dimension elements into their queries, you can also specify a DimensionElement object when instantiating a QueryCategory. To specify a **COMPARE** or **BUCKET** expression as a QueryCategory, see "Extension Functions as QueryCategory Expressions" on page 5-42.

Figure 8-5
QueryCategories Class of Objects: Properties

Properties	Description/Example
Category	This property stores a string value representing the name of the attribute or the syntax of the expression that defines groupings within a query. Assigning a new value to this property alters the grouping of the query originally specified when the QueryCategory is instantiated. This property is the default property of the **QueryCategories** class of objects. `MsgBox MyQueryCategory.Category`
Name	This property stores a label identifying the QueryCategory. Changing the value of this property leaves the grouping of the query unaltered but changes the label MetaCube returns when displaying QueryCategory values. This property is unlike the **BUCKET** function, which groups and labels values of the QueryCategory rather than the QueryCategory itself. Until a new label has been assigned to the QueryCategory, this property stores the same string value as the Category property. `Let MyQueryCategory.Name = "NewName"`
Object	This read-only property points to the Attribute object on which a QueryCategory expression is based. The QueryCategory object often represents an attribute object, and in such cases this property points to an object similar to the QueryCategory itself. But when a QueryCategory object represents a bucket or some other expression, the Object property points to the Attribute object to which the syntax of that bucket expression refers. The Object property of a QueryCategory representing buckets on the **Brand** attribute would, for example, point to the Attribute object for **Brand**. This property stores a null value for expressions based on multiple Attribute objects, such as a comparison. `MsgBox MyQueryCategory.Object.Name`

(1 of 2)

Properties	Description/Example
Orientation	This long constant determines whether each of the values returned by the QueryCategory object will be displayed in separate rows, columns, or pages. By default, MetaCube displays such values in the row orientation. Changing the value of this property after a report has been generated ultimately involves instantiating a new MetaCube object but does not require the MetaCube analysis engine to requery the database. Figure 8-6 lists the appropriate constants. `MyQueryCategory.Orientation = OrientationColumn`
Parent	Stores the Query object to which the QueryCategory object belongs. `MyQueryCategory.Parent.Name`
SortDirection	This long constant determines whether values of an attribute are sorted in ascending (alphabetical) or descending (reverse-alphabetical) order. Changing the value of this property after a report has been generated ultimately involves instantiating a new MetaCube object but does not require the analysis engine to requery the database. See Figure 8-6 on page 8-24 for a listing of numeric arguments and their corresponding constants. The **SortRow** and **SortColumn** properties of the MetaCube object class, which sort columns or rows on the basis of a measure's values, over ride conflicting sorts applied by this property. See "Exercise 23: Sorting" on page 8-54. Defaults to ascending order. `MyQueryCategory.SortDirection = SortDirectionDesc`

(2 of 2)

Figure 8-6 summarizes the constants for the QueryCategory object's **Orientation** property.

Figure 8-6
Constants for the QueryCategories Object Class Orientation Property

Orientation	MetaCons.bas Constant Name	Constant
Rows	OrientationRow	1
Columns	OrientationColumn	2
Pages	OrientationPage	3

The SortDirection Property

Sorts allow you to arrange values within an attribute in alphabetical or reverse-alphabetical order and to arrange values of a measure in descending or ascending order of magnitude. For more information about sorting, consult the *MetaCube Explorer User's Guide*.

By default, all attribute values are sorted in ascending, or alphabetical order, whereas measures are not sorted at all. To sort measures, assign a value to the **SortColumn** or **SortRow** properties of the MetaCubes class of objects, identifying the column or row of numeric data on which you want to perform the sort. See Figure 8-17 on page 8-47.

Figure 8-7 summarizes the numeric values for the **SortDirection** property.

Figure 8-7
Numeric Constants for the SortDirection Property

Sort Direction	MetaCons.bas Constant Name	Constant
No Sort	SortDirectionIgnore	0
Ascending Order	SortDirectionAsc	1
Descending Order	SortDirectionDesc	2

The QueryItems Class of Objects

Just as the QueryCategory object identifies an attribute to apply to a query, the QueryItem object identifies a measure to apply to a query.

To include a measure in a query's definition, simply instantiate a QueryItem object, identifying in the **Add** method's argument the precise name of the Measure object:

```
MyQuery.QueryItems.Add "Units Sold"
```

Because you can perform calculations on and apply formats to a measure, the QueryItem object's properties are slightly more extensive than the properties for a QueryCategory object.

QueryItems Properties

The **QueryItems** object class has several properties, summarized in Figure 8-8.

Figure 8-8
QueryItems Class of Objects: Properties

Properties	Description/Example
FormatString	String. Stores syntax for formatting numeric data in the application or control responsible for displaying query results. See "The FormatString and FormatStrings Properties: An Overview" on page 8-27. `MyQueryItem.FormatString = "#,##0.00"`
Item	String. Stores a value representing the name of the measure or the syntax of the expression that defines a QueryItem. Assigning a new value to this property alters the numeric data retrieved by the query originally specified when the QueryItem is instantiated. This property is the default property of the **QueryItems** class of objects. `MsgBox MyQueryItem.Item`

(1 of 2)

Properties	Description/Example
Name	This property stores a label identifying the QueryItem in a report. Changing the value of this property leaves the numeric data of the query unaltered but changes the label MetaCube returns when displaying the name of the QueryItem. Until a new label has been assigned to the QueryItem, this property stores the same string value as the **Item** property. `Let MyQueryItem.Name = "NewName"`
Object	Read-only. Stores the Measure object on which a QueryItem expression is based. The QueryItem object often itself represents a Measure object, and in such cases, this property stores an object similar to the QueryItem. But when a QueryItem object represents an expression such as **MovingAvg** or **TOPN**, the Object property stores the Measure object to which the syntax of the expression refers. This property would store a null value for expressions based on multiple Measure objects, although no such expressions currently exist. `MsgBox MyQueryItem.Object.Name`
Parent	Object. Returns the Query object to which the collection of **QueryItems** belongs. `MsgBox MyQueryItem.Parent.Name`

(2 of 2)

The **QueryItems** object class features no methods and owns no collections.

The FormatString and FormatStrings Properties: An Overview

Properties of MetaCube, QueryItem, and Measure object classes are responsible for storing the syntax for formatting numeric data in different display environments, such as an Excel Spreadsheet or a reporting control. The **FormatStrings** property of the **MetaCubes** object class is documented in Figure 8-17 on page 8-47, and the **FormatString** property of the Measure object is documented in Figure 6-8 on page 6-24.

Both the **FormatString** property of the QueryItem object and the same property of the Measure object define the syntax for a measure's formatting, which the MetaCube analysis engine can store but not interpret or process. An object control or the application itself, and not MetaCube, ultimately formats the data by interpreting the formatting syntax stored by MetaCube in MetaCube's metadata or, in the case of a QueryItem, in memory. The syntax of the string thus varies, depending on the formatting information required by the object control or by the application to format numeric data.

Although the **FormatString** property of the **QueryItems** object class defines the formatting of a measure as it will be displayed for a particular query, the same property of the **Measures** object class assigns a default format to the measure for all queries. The property of the **QueryItems** object class can be assigned prior to running a query, overriding the property of the **Measures** object class, set when defining MetaCube's metadata.

The **FormatString** properties of the **QueryItems** and **Measures** object classes enable you to define formatting syntax, but the **FormatStrings** property of the **MetaCubes** object class returns such syntax in a read-only ValueList, providing a single, convenient source on which the reporting application or control can rely for all formatting syntax.

For any given query, the MetaCube object's **FormatStrings** property stores the format for each of the measures included in that query, regardless of whether that format was assigned when defining a query or defaults from metadata specifications. The formatting commands are repeated in the ValueList for each column or row of a given measure appearing in the report, alternating, if necessary, with any other measures that might also be included in the report. An array of formatting syntax organized in the same order as the columns of numeric data in the report enables you to easily apply different formats to different measures, even in a report with many columns or rows of numeric data, as shown in Exercise 22.

You should thus use:

- the **FormatString** property of the **Measures** object class when defining default formats for a measure in MetaCube's metadata.
- the same property of the **QueryItems** object class when specifying a different format for the purposes of a single query.
- the **FormatStrings** property of the MetaCube object to determine the formats that have ultimately been assigned to all measures included in a query.

The syntax included in the example is Excel-compliant, as is the syntax displayed in MetaCube Explorer. The pound sign (#) indicates the places where a digit may be displayed; a zero (0) indicates places where a zero may appear if a digit does not exist. Commas typically demarcate every three decimal places; and periods indicate the position of the decimal point. For more information, consult Microsoft Excel documentation or review the syntax appearing in the Format Cells dialog box, which can be opened by choosing **Format→Cells** in any active worksheet.

Exercise 22 incorporates the **FormatString** and **FormatStrings** properties of all three object classes to enable users to review a measure's default formatting, as defined in MetaCube's metadata, and to suggest a new format. The query includes an attribute as well as two measures organized by columns, demonstrating the usefulness of the **MetaCubes** object class's **FormatStrings** property in relatively complex queries.

Exercise 22: Formatting Measures

```
Option Explicit

'MetaCube API Exercise 22: Formatting Measures

Sub MetaCube_API()

'Declare Variables
    Dim MyMetabase As Object, MyQuery As Object, _
        MyFilter As Object, MyMetaCube As Object, _
        MyData As Variant, ExcelFormat As String, _
        FormatArray As Variant, Count As Integer

'Excel Variables
    Dim ReportRange As Range

    Const OrientationColumn = 2
'Connect
    Set MyMetabase = CreateObject("Metabase")

    'Identify an ODBC Data Source
    Let MyMetabase.ConnectString = "Metademo"

    'Specify a set of metadata
    Let MyMetabase.Name = "MetaCube Demo"

    'Specify login to database
    Let MyMetabase.Login = "metademo"
    'Passes value to database on connection

    'Specify database password
    Let MyMetabase.Password = "Metademo"

    MyMetabase.Connect

'Define Query
    Set MyQuery = _
        MyMetabase.Queries.Add("My New Query")
    MyQuery.QueryCategories.Add "Region"
    MyQuery.QueryCategories.Item("Region") _
        .Orientation = OrientationColumn 'More columns
    MyQuery.QueryCategories.Add "Brand"
    MyQuery.QueryItems.Add "Net Profit" 'New format
      MyQuery.QueryItems.Add "Units Sold" 'Default format

'Prompt for new format of Net Profit measure...
    Let ExcelFormat = InputBox _
    (Title:="Format Measure Dialog", _
```

```
48        Prompt:= _
49         "Profits currently displayed in " _
50         + MyMetabase.FactTables.Item _
51        ("Sales Transactions").Measures.Item _
52        ("Net Profit").FormatString + _
53        " format.  Enter a new format for profits:")
54      Let MyQuery.QueryItems.Item _
55        ("Net Profit").FormatString = ExcelFormat
56  'Get Results
57      Worksheets.Item("Query Report").Activate
58      Cells.Select
59      Selection.ClearContents
60
61      Set MyMetaCube = MyQuery.MetaCubes.Add("Data")
62      Let MyData = MyMetaCube.ToVBArray
63      Set ReportRange = ActiveSheet.Range _
64          (ActiveSheet.Cells(1, 1), _
65          ActiveSheet.Cells _
66          (MyMetaCube.Rows, MyMetaCube.Columns))
67      Let ReportRange.Value = MyData
68      ReportRange.EntireColumn.AutoFit  'Size columns
69
70  'FormatData
71      Let FormatArray = _
72        MyMetaCube.FormatStrings.ArrayValues 'Get formats
73      'There's an array value for each measure-column; cycle
    through…
74      For Count = 1 To UBound(FormatArray)
75          Columns(Count + 1).Select
76          Selection.NumberFormat = FormatArray(Count)
77      Next Count
78
79  End Sub
```

Explanation of Exercise 22

This procedure defines the familiar **Brand-Region** query, pivoting the **Region** attribute to the column orientation, so that the two measures included in the query, **Net Profit** and **Units Sold** appear twice, once for each region. Depending on the orientation of measures, the **FormatStrings** property of the MetaCube object returns an array listing the appropriate formatting for each column or row in the report, in the order in which the different measures are displayed in the report.

After it defines a simple query, the procedure prompts the user to enter a new format for the **Net Profit** measure on lines 45 to 55. Embedded in the prompt at lines 50 to 52 is the default formatting syntax for the measure, as defined in MetaCube's metadata. The **FormatString** property of the **Measures** class of objects can be assigned new values when creating or editing metadata.

To avoid an extremely long line of code, the user's input is stored in a variable, **ExcelFormat**, and subsequently assigned to the **FormatString** property of the QueryItem representing **Net Profit** in lines 54 and 55. Because you assign a new value to the QueryItem's **FormatString** property rather than the Measure object's **FormatString** property, the new format applies only to the query including that QueryItem.

The query subsequently executes, returning the data into a spreadsheet named **Query Report**. As usual, a spreadsheet of this name must exist before executing the procedure.

After the query has executed and the data returns to a range within a spreadsheet, one task remains for the MetaCube analysis engine: to communicate the format for each measure to Excel, where the formatting is ultimately performed. The **FormatArray** variable stores an array of formatting syntax, with a separate string for each column of numeric data in the report, indicating how that column should be formatted. In this procedure, the syntax alternates between the new formatting for profits, and the default formatting for sales. MetaCube generates this array of alternating syntax, holding the array as a ValueList in **MyMetaCube**'s **FormatStrings** property.

Since the **FormatArray** variable contains a separate value for each column in the report, we can deploy Visual Basic for Application's **Ubound** function to determine the number of values in the array and thus the number of columns storing numeric data in the report, as shown on lines 71 and 72. Using a For… Next loop, the procedure iteratively formats each column. The loop counter, represented by the variable **Count**, serves two purposes:

- Identifying the column to format on line 71
- Locating a value within the array on line 72.

  This value contains the appropriate formatting syntax for the selected column.

The formatting begins with the second column of the report, which actually contains the first column of numeric data. For this reason, the **Count** variable is incremented by one when identifying the column to select.

The order of instantiation of different QueryCategory objects determines the order in which MetaCube groups values. If, for example, you instantiate a QueryCategory identifying the **Brand** attribute first and one identifying a **Region** attribute second, regions will be grouped by brand as long as those values are either both organized by column or both organized by row. You can re-arrange this configuration by deploying the **MakeFirst** method on an item within the **QueryCategories** collection.

The Filters Class of Objects

Filters limit the range of data retrieved by a query or incorporated into an aggregate table. To apply a filter to the data represented by a particular object, you must either include that filter in the collection of filters owned by that particular object or identify that filter through one of the object's properties, as required.

The Filters Collection's Methods

The **Filters** object collection features its own set of methods, summarized in Figure 8-9.

Figure 8-9
Filters Collection: Methods

Method	Description/Example
AddDefault	Applies the default filter for a particular dimension to a query. You can deploy this method only when adding a Filter object to a collection owned by a query, and you must specify as an argument the name of the Dimension object for which you are applying a default filter. Each Dimension object features the **DefaultFilter** property, which identifies a particular Filter object as that dimension's default. `MyQuery.Filters.AddDefault "Time"`
AddNewFilter	Adds a new filter. You can deploy this method to create a new filter, either on an ad hoc basis for a particular query or to save that filter in MetaCube's metadata. To instantiate a new Filter object, you must specify its name as an argument to this method. Once you have created the filter, you can add FilterElement objects to define the criteria by which the filter limits data. `MyQuery.Filters.AddNewFilter "This Month"`

AddSaved	Retrieves the definition of a filter stored in MetaCube's metadata tables from a library of filters associated with a particular folder and applies that filter to the Query object owning the **Filters** collection. Although MetaCube Explorer prevents a user from deploying others' private filters, this security measure is artificially imposed by the application, not the programming interface. Through the programming interface, you can apply any filter in the DSS System to a query. The **AddSaved** method requires three arguments: the name of the saved filter; the name of the user who originally created the saved filter; and the folder, as an object, in which the filter is stored. To retrieve a public filter, specify the user name **metapub**.
	`MyQuery.Filters.AddSaved "This Month", "MetaDemo", _`
	`MyMetabase.RootFolder`

(1 of 2)

Method	Description/Example
CountGroup	Returns as an integer the number of filters associated with a particular dimension or group. You must specify as an argument the name of the Dimension object by which filters are usually grouped. Filters may also be grouped by a fact table, in which case the group is named after the fact table (MetaCube Explorer prefixes fact table groups with the syntax `FactTables`). If the filters were created in a custom application, they may be grouped using some other logic. `MsgBox MyQuery.Filters.CountGroup "Time"`
ItemGroup	Retrieves a Filter object from the subset of filters associated with a particular dimension or group and identifies the object within that subset by an index number. This allows you to specify, for example, the second saved filter associated with the Time dimension. You must specify the name of the Dimension object by which the filters are grouped and the index number of the particular item within that group. `Set MyFilter = MyQuery.Filters.ItemGroup("Time", 1)`
Remove	Stops the application of a filter to a particular query without deleting the filter definition from the metadata. You must specify as an argument the index number of the filter within that group. `MyQueries.Filters.Remove 2`

(2 of 2)

Filters Properties

The Filter object's properties describe the general characteristics of a filter, such as its name, its owner, and the dimension with which it is associated. A Filter object collection of **FilterElement** objects defines the actual constraint or criterion by which the filter limits a range of data. Figure 8-10 summarizes the general properties of the Filter object.

Figure 8-10
Filters Class of Objects: Properties

Property	Description/Example
Enabled	Boolean. Setting the value of this property to `False` precludes MetaCube from applying the filter to the query result without excluding the filter from the collection of filters owned by the Query object. Defaults to `True`. `MyFilter.Enabled = False`
Folder	Object. Read-only. Represents the Folder object under which the Filter object has been saved. This property is invalid for Filter objects that have not been saved. `MsgBox MyFilter.Folder.Name`
Group	String. Identifies the name of the Dimension object or FactTable object by which the filter is grouped. You can also specify a new criterion, corresponding to either Dimension or FactTable names, by which filters are grouped. Although the programming interface allows you to define a filter constraining values of measures, attributes and dimension elements from different dimensions or fact tables, this property associates each Filter object with a particular dimension or fact table. This association enables MetaCube Explorer and other applications to organize filters by dimension or fact table for display in a user interface. `MyFilter.Group = "Time"`
Name	String. Identifies the Filter object. The default property. `MsgBox MyFilter.Name`
Owner	String. Read-only. Identifies the name of the user who owns the filter. Defaults to the user name provided at login. `MsgBox MyFilter.Owner`

(1 of 2)

Property	Description/Example
Parent	Object. Query object. `MsgBox MyFilter.Parent.Name`
Saved	Boolean. Indicates whether the definition of the filter changed since its most recent save. This read-only property returns `False` if the filter has changed, `True` otherwise. `MsgBox MyFilter.Saved`
Updatable	Boolean. Indicates whether the current user has privileges to edit the filter. `MsgBox MyFilter.Updatable`

(2 of 2)

Filters Methods

The Filter object's methods save and delete Filter objects. In saving a Filter object, you store the definition of the filter in metadata tables on the relational database. In deleting a filter, you destroy the database records storing that filter's definition. Figure 8-11 summarizes the Filter object's methods.

Figure 8-11
Filters Class of Objects: Methods

Method	Description/Example
DeleteFilter	Deletes the metadata storing a filter's definition in the relational database, regardless of whether the corresponding Filter object exists within a collection owned by a Query object or a Metabase object. When you delete a Filter object, MetaCube not only no longer recognizes the filter's existence, the definition of the filter no longer exists. This method requires no arguments. `MyFilter.DeleteFilter`
FullPathName	Returns the full path to a Filter object, which may be necessary to identify a mandatory filter. Typically mandatory filters are assigned using MetaCube Secure Warehouse. You must specify as an argument the name of the Filter object for which a full path is needed. `UserFilterPath = MyFilter.FullPathName("UserFilter")`
Save	Saves the definition of an existing filter in metadata tables on the relational database. The filter is saved under its previous name, folder, owner, and group. To change any of these values, deploy the **RenameFilter** method of the **Folders** object class. To create a copy of a Filter with a new name, folder, or group, deploy the **SaveAs** method of the **Filters** object class. Both methods are illustrated in "Exercise 19: Saving Queries and Filters to Folders; Renaming, Opening Queries and Filters from Folders" on page 7-7. `MyFilter.Save`
SaveAs	Saves a new Filter object or saves an existing Filter object under a new name or in a different folder. Before saving the filter, the group to which the filter belongs must be specified by assigning a value to the **Group** property of the Filter object. For this method, the name of the filter is specified as a string argument, and the folder into which the definition should be stored as an object: `MyFilter.SaveAs "My Brands", MyMetabase.RootFolder` See "Exercise 19: Saving Queries and Filters to Folders; Renaming, Opening Queries and Filters from Folders" on page 7-7 for an example of an application involving this method.

The FilterElements Class of Objects

A filter consists of one or more criteria limiting the range of a data set in a report or an aggregate. The criteria defining a filter are represented as a collection of FilterElement objects owned by the corresponding Filter object. Just as filters can be applied to a query or to an aggregate, so a FilterElement object can be applied directly to a measure as a constraint. For an explanation of measure constraints, which preclude a calculated measure from performing operations like division by zero, see "The Measures Class of Objects" on page 6-22. For an introduction to aggregate filters, see Figure 6-3 on page 6-11. This section discusses how FilterElements limit the data returned by a query.

Each FilterElement object identifies values within a measure, attribute, or dimension element to include or exclude from a report. A criterion may, for example, include the East value of the **Region** attribute, such that a query can retrieve only transactions associated with that region. A query definition need not group transactions by the **Region** attribute to apply a filter on the **Region** attribute against that query.

In the previous example, we defined a constraint that chose all transactions for which the **Region** attribute equaled East. You can also define constraints incorporating more sophisticated operators, such as <, >,<>, <=, >=, like, in, and not null. For the numerical values of a measure or a dimension element these operators can perform functions like including only sales greater than $1,000 or stores with codes less than 800.

The less-than and greater-than operators can also apply alphabetically to the string values of an attribute, with a less-than operator corresponding to earlier in an alphabetical list of values and a greater-than operator corresponding to later in the alphabetical list of string values. The in operator enables you to include multiple, explicitly identified values in a constraint, such as the East region and the West region. The like operator includes all values containing a set of letters or digits, with the % symbol demarcating either end of the set.

Because MetaCube recognizes the date of the most recent transactions stored in the data warehouse, you can define dynamic, time-dependent filters, which include different values depending on which values in the data warehouse are most recent. For example, you can limit data to the most recent three weeks of transactions or to transactions for this month and the same month last year.

Rather than specifying a value of the attribute or dimension element on which you are filtering, you must specify one of the following dynamic parameters: Current Period, Last N Periods (where N can be any integer), Current Period and Same Period Last Year, or Same Period Last Year. The duration of the period corresponds to the type of time attribute or dimension element on which you are filtering, be it Day, Week, Month, Quarter, and so forth. When specifying a parameter, you must always enclose the parameter in angle brackets, so that MetaCube does not mistake the parameter for an actual value, such as `<<Current Period>>` or `<<Last 4 Periods>>`.

Although a properly configured MetaCube system can recognize such common relative-time parameters, you can also create new parameters, prompting users to enter the values to be substituted for these parameters immediately prior to execution of the query. Such parameters can be assigned any label, as long you bracket that label within less-than and greater-than operators. It is convenient to label the parameter such that it may subsequently be offered as a prompt in a dialog box requesting the values of the parameter. See "Exercise 20: Executing Queries That Include Parameterized Filters" on page 8-14.

FilterElements Properties

Different properties of the FilterElement object identify each component of a constraint. Figure 8-12 summarizes the properties of the FilterElement object.

Figure 8-12
FilterElements Class of Objects: Properties

Property	Description/Example
FilterOn	String. Identifies the name of the attribute, dimension element, or measure from which the values are to be selected in the constraint. `MyFilterElement.FilterOn = "Week"`

FilterType	Long. Read-only. Indicates whether the FilterElement is based on an attribute, measure, or dimension element object. For the correspondence between the numeric value stored by the property and the filter type, see Figure 8-13 on page 8-42.
	`If MyFilter.FilterType = FilterTypeStandard Then MsgBox _`
	`"Attribute-based filter"`

Object	Object. Represents the attribute, measure, or dimension element object on which the FilterElement object is based. `MsgBox MyFilterElement.Object.Name`
Operand	String. Identifies the particular values within the attribute, dimension element, or measure on which to perform an operation for the constraint. Parameters should be enclosed in less-than and greater-than operators. `MyFilterElement.Operand = "<<Current Period>>"`
Operator	String. The operator within the constraint definition. Any operator that can be included in the WHERE clause of an SQL statement is acceptable, including the following: =, <, >,<>, <=, >=, in, not in, like, not like, is null, and is not null. The in operator requires **Operand** values to be enclosed in parentheses and separated by commas, as shown in the example for instantiating a FilterElement object. `MyFilterElement.Operator = "="`
Parent	Object. Filter object. `MsgBox MyFilterElement.Parent.Name`

To instantiate a FilterElement object, you must specify as arguments values for the **FilterOn**, **Operator**, and **Operand** properties:

```
MyFilter.FilterElements.Add "Brand", "In", "('Alden',
'Delmore')"
```

The FilterElement object does not feature any properties, nor does it own any collections.

Figure 8-13 summarizes constants stored by the **FilterType** property of the FilterElement object:

Figure 8-13
FilterType Constants

Filter Element Type	MetaCons.bas Constant Name	Constant
Other	FilterTypeOther	0
Attribute	FilterTypeStandard	1
Dimension Element	FilterTypeDimensionElement	2
Measure	FilterTypeMeasure	3

The MetaCubes Class of Objects

A MetaCube object can best be thought of as a virtual cube of data, as returned by a Query object. The MetaCube object is typically described as if it actually stores data in a physical cube-like structure. In actuality, MetaCube avoids the expansion problems and lack of sparsity associated with storing data in this format but handles data as though it were.

Each MetaCube object within the collection owned by a query represents a different way of organizing, summarizing, and pivoting data. The data from a given query, for example, can be presented in a break report, a cross-tabular report, or even organized on different pages with results subtotaled or averaged. Each new format or calculation manipulates the same set of data returned to the client, and each corresponds to a different MetaCube object within a Query object's collection.

Instantiating a MetaCube Object

Although a MetaCube object ultimately represents the structure storing and manipulating the data returned by a query, you can instantiate a MetaCube object before the query executes, including the name of the object as the only arguments to the Query object's **Add** method:

```
MyQuery.MetaCubes.Add "My Cube of Data"
```

If you add multiple instances of a MetaCube object to a collection owned by a single query, the query does not return a separate data set for each MetaCube object. Rather, each instance organizes and manipulates a common set of data in a different way.

General Properties

A standard set of MetaCube object properties describe the MetaCube object generally, identifying the name of the object and its parent. Included in this set are Boolean properties that indicate whether the analytical engine performs grand total calculations on the report represented by the object and whether duplicate values are displayed in break reports.

As an object representing a virtual three-dimensional data structure defined by columns, rows, and pages, the MetaCube object also features properties that:

- describe a particular three-dimensional location within a cube.
- retrieve information about a particular cell within the cube.
- describe the general three-dimensional structure of a cube.

For example, before determining the value of a cell using the **Cell** property, you must identify the location of the cell using the **Column**, **Row**, and **Page** properties.

Any MetaCube property or method representing or returning a value that depends on the nature of the data to be retrieved by the query can force the query to execute before or even without deploying that Query object's **Retrieve** method. The **Rows**, **Columns**, **Pages**, **Cell**, and **CellType** properties, as well as the **FetchCell**, **FetchCellType**, **ToSpreadClip**, and **ToVBArray** methods, all implicitly require a query to execute. Once a query has executed, however, none of these methods require the query to re-execute. For an example of an application that relies on a method of the MetaCubes object class to execute a query, see Exercise 3 on page 2-13.

Figure 8-14
MetaCubes Class of Objects: General Properties

Property	Description/Example
Name	String. The name of the MetaCube object, as specified upon instantiation. Default property. `MsgBox MyMetaCube.Name`
Parent	Object. Stores the Query object. `MsgBox MyMetaCube.Parent.Name`
Scratch	Boolean. `False` indicates that the report definition, as represented by the MetaCube object, are saved with the query definition. The default value, `True`, indicates that the format of a query's report, including pivoting, subtotals, and sorts are not saved with the query's definition. `MyMetaCube.Scratch = False`
Suppress-Duplicates	Boolean. In a report including rows and subrows or columns and subcolumns, the values in a row or a column group values in subrows or subcolumns. If you want the row or column values duplicated for each value within the subrow or subcolumn, set the **SuppressDuplicates** property to `False`. See Figure 8-15 on page 8-45 and Figure 8-16 on page 8-46. To suppress the duplicate values in grouping rows or columns, set this property to `True`. Defaults to `False`. `MyMetaCube.SuppressDuplicates = True`

Figure 8-15 displays the results of a query with the **SuppressDuplicates** property of the MetaCube object set to `False`.

Figure 8-15
Brand-Region Query, SuppressDuplicates Property False

Brand	Region	Units Sold
Alden	Northeast	1811
Alden	West	2626
Barton	Northeast	1314
Barton	West	1924
Delmore	Northeast	1778
Delmore	West	2557
Extreme	Northeast	433
Extreme	West	649
Lasertech	Northeast	1105
Lasertech	West	1665
NVD	Northeast	2719
NVD	West	3788
Onetron	Northeast	910
Onetron	West	1254
Suresound	Northeast	2548
Suresound	West	3464
Techno Components	Northeast	3699
Techno Components	West	5286

Figure 8-16 displays the results of a query in which the **SuppressDuplicates** property of the MetaCube object has been set to `True`.

Figure 8-16
Brand-Region Query, SuppressDuplicates Property True

Brand	Region	Units Sold
Alden	Northeast	1811
	West	2626
Barton	Northeast	1314
	West	1924
Delmore	Northeast	1778
	West	2557
Extreme	Northeast	433
	West	649
Lasertech	Northeast	1105
	West	1665
NVD	Northeast	2719
	West	3788
Onetron	Northeast	910
	West	1254
Suresound	Northeast	2548
	West	3464
Techno Components	Northeast	3699
	West	5286

Properties of the Three-Dimensional Virtual Cube

Figure 8-17 summarizes the properties describing three-dimensional characteristics of the MetaCube object.

Figure 8-17
MetaCubes Class of Objects: Properties for Identifying Cells in a Virtual Three-Dimensional Space and Sort-Related Properties

Property	Description/Example
Cell	Variant. Returns the value of a cell within the MetaCube object's virtual cube of data. The particular cell is specified by a set of **Row**, **Column**, and **Page** properties. This property represents the same value retrieved by the MetaCube object's **FetchCell** method. Either the property or the method can implicitly require a query to execute, if necessary. Read-only. `MsgBox MyMetaCube.Cell`
CellError	Double. For query results extrapolated from a sample table, this property stores the range of error associated with the value of a particular cell. The range varies directly with the confidence with which MetaCube is required to certify that the actual result falls within that range. As with the **Cell** property, the particular cell for which MetaCube evaluates the error is specified by a set of **Row**, **Column**, and **Page** properties. This property represents the same value retrieved by the MetaCube object's **FetchCellError** method. For a complete discussion of sampling, see "The Samples Class of Objects" on page 6-30. This property is read-only. `MsgBox MyMetaCube.CellError`
CellType	Integer. Identifies the type of data in a specified cell. Different integer values correspond to an attribute / dimension element value, a measure value, a subtotal or grand total value, an attribute / dimension element label, a measure label, or a subtotal or grand total label. Figure 8-18 on page 8-53 summarizes the significance of each cell type code. This property is read-only. The type of each cell depends on the definition of the query and the format of the report. `MsgBox MyMetaCube.CellType`

(1 of 6)

Property	Description/Example
Column	Long integer. Identifies a particular column in the virtual cube of data. Defaults to 0. `MyMetaCube.Column = 1`
Columns	Long integer. Returns the number of columns in the virtual cube of data represented by the MetaCube object. See the tutorial application developed in the second chapter of this reference, which defines a range of cells in which to display a result using the MetaCube object's **Columns** and **Rows** properties. The number of columns in a virtual cube of data depends on the number of attributes, measures, and dimension elements included in a query and the range of values each represents. This property is read-only. `MsgBox MyMetaCube.Columns`
Current-Attribute	QueryCategory object. Represents the attribute / dimension element to which the currently selected or identified attribute / dimension value belongs. `MsgBox MyMetaCube.CurrentAttribute.Name`
FormatStrings	ValueList. An array of formatting commands for each numeric column or row in the query, as set by the **FormatString** property of the QueryItems and Measures object classes. See "The SortDirection Property" on page 8-25. `MsgBox MyMetaCube.FormatStrings`
Page	Long integer. Identifies a particular page in the virtual cube of data. Defaults to 0. `MyMetaCube.Page = 1`
PageLabels	ValueList. This property stores the names of attribute values appearing in the page orientation for a given page, and is defined by the **Page** property of the MetaCube object. The ValueList contains multiple values only if more than one QueryCategory has been pivoted to a page orientation. `MsgBox MyMetaCube.PageLabels.TabbedValues`
Pages	Long integer. Represents the number of pages in the virtual cube of data represented by the MetaCube object. `MsgBox MyMetaCube.Pages`

(2 of 6)

Property	Description/Example
Row	Long integer: Identifies a particular row in the virtual cube of data represented by the MetaCube object. Defaults to `0`. `MyMetacube.Row = 1`
Rows	Long integer. Returns the number of rows in the virtual cube of data represented by the MetaCube object. The number of rows depends on the definition of the query and the number of records retrieved by the query. This property is read-only. `MsgBox MyMetaCube.Rows`
SortColumn	Long. Indicates the column of numeric data on which to perform a sort, organizing rows according to the numeric value of the measure appearing in each row. Columns are numbered from left to right, beginning with 0 at the first column in the report. Pivoting measures to the row orientation does not alter the role of this property. Using this property to identify a column of attribute values returns an error. Assigning a value to this property overrides any sort applied to a QueryCategory organized by rows, because this method sorts rows based on the values of a measure rather than the values of an attribute. Unlike the QueryCategory object's **SortDirection** property, which sorts attribute values regardless of their position in the report, this property offers developers a powerful way to apply a new sort to an arbitrary numeric column within an existing report. Specifying the column on which to base a sort of rows prior to processing the query can be difficult, as the order in which columns appear depends on the query result. Defaults to `-1`, directing MetaCube to sort reports based on the **SortDirection** properties of the Query object's collection of QueryCategories, as documented in "The SortDirection Property" on page 8-25. See also Exercise 23 on page 8-54. In cases where MetaCube sorts numeric data on rows and on columns before the query executes, rows are sorted first. `Let MyMetaCube.SortColumn = 2`

(3 of 6)

Property	Description/Example
SortColumn-Breaks	Boolean. Indicates how to sort reports in which more than one QueryCategory object has a row orientation. A `True` value directs MetaCube to perform a numeric sort only on subrows, leaving rows unsorted. In this case, only the rows within each break are sorted from small to large numbers or vice-versa. A `False` value directs MetaCube to sort the entire report so that the smallest numbers appears first and the largest last or vice versa, regardless of the break to which a subrow belongs. Defaults to `False`. For examples, see Figure 8-21 on page 8-59. `Let MyMetaCube.SortColumnBreaks = True`
SortColumn-Direction	Long. Indicates the direction in which the column specified by the **SortColumn** property is sorted. An ascending sort arranges numbers such that the largest numbers appear at the bottom of the report and the smallest numbers at the top. Defaults to ascending order. See Figure 8-7 on page 8-25 for a list of constants declarations, their significance, and their corresponding numeric values. `MyMetaCube.SortColumnDirection = SortDirectionDesc`

(4 of 6)

Property	Description/Example
SortRow	Long. Indicates the row of numeric data on which to perform a sort. Columns are organized according to the numeric value of the measure appearing in each column. Rows are numbered from top to bottom, beginning with 0 at the first row in the report. Using this property to identify a row of attribute values returns an error. Assigning a value to this property overrides any sort applied to a QueryCategory organized by columns because this method sorts columns based on the values of a measure rather than the values of an attribute. Since QueryCategories are sorted before numeric data, changing the sort on a QueryCategory object organized by rows obviously changes the values that appear in any given row specified by this property. Like the **SortColumn** property of the MetaCube object, this property offers developers a powerful way to apply a new sort to an existing report. Specifying the row on which to base a sort of columns prior to processing the query can be difficult, as the order in which rows appear depends on the query result. Defaults to `-1`, which directs MetaCube to sort reports based on the **SortDirection** properties of the Query object's collection of QueryCategories, as documented in "The SortDirection Property" on page 8-25. See also Exercise 23 on page 8-54. In cases where MetaCube sorts numeric data on rows and on columns before the query executes, rows are sorted first. `Let MyMetaCube.SortRow = 2`

(5 of 6)

Property	Description/Example
SortRowBreaks	Boolean. Indicates how to sort reports in which more than one QueryCategory object has a column orientation. A `True` value directs MetaCube to perform a numeric sort only on subcolumns, leaving columns unsorted. In this case, only the columns within each break are sorted from small to large numbers or vice versa. A `False` value directs MetaCube to sort the entire report so that the smallest numbers appear first and the largest last or vice versa, regardless of the break to which a subcolumn belongs. Defaults to `False`. For examples, see Figure 8-21 on page 8-59. `MyMetaCube.SortRowBreaks = True`
SortRow-Direction	Long. Indicates the direction in which the row specified by the **SortRow** property is sorted. An ascending sort arranges numbers such that the largest numbers appear in columns at the right of the report and the smallest numbers in columns at the left. Defaults to ascending order. See Figure 8-7 on page 8-25 for a list of constants declarations, their significance, and their corresponding numeric values. `MyMetaCube.SortRowDirection = SortDirectionDesc`
Value	Variant. Returns the value of a specified cell. This property is identical to the **Cell** property but included for compatibility with Visual Basic controls. `MsgBox MyMetaCube.Value`

(6 of 6)

Related Numeric Constants

Figure 8-18 explains the numeric values returned by the **CellType** property and their associated constants.

Figure 8-18
MetaCubes Class of Objects: Numeric Values for The CellType Property

Cell Contents	MetaCons.bas Constant Name	Constant
Empty	CellTypeEmpty	0
QueryCategory name	CellTypeCategoryLabel	1
QueryCategory value	CellTypeCategoryValue	2
Label for column or row containing calculated information	CellTypeSummaryLabel	3
Calculated data, such as subtotals or grand totals	CellTypeSummaryValue	4
Measure name	CellTypeQueryItemLabel	5
Numeric measure value	CellTypeQueryItemValue	6

Sorting: SortDirection and SortColumn Properties

Sorting a standard query in different ways and iteratively executing reports demonstrates MetaCube's complex sorting features, as shown in Exercise 23. Before executing the procedures in this exercise, you must create three spreadsheets with the names **Sort on Brand**, **Sort on Measure**, and **Sort on Brand Again** because the **Sort** procedure attempts to create reports in sheets with these names.

Exercise 23: Sorting

```
Option Explicit

'MetaCube API Exercise 23: Sorting

Sub MetaCube_API()

'Declare Variables and Constants
Dim MyMetabase As Object, MyQuery As Object, _
    MyMetaCube As Object, MyData As Variant
Const OrientationColumn = 2
Const SortDirectionDesc = 2

'Login
Set MyMetabase = CreateObject("Metabase")

'Identify an ODBC Data Source
Let MyMetabase.ConnectString = "Metademo"

'Specify a set of metadata
Let MyMetabase.Name = "MetaCube Demo"

'Specify login to database
Let MyMetabase.Login = "metademo"
'Passes value to database on connection

'Specify database password
Let MyMetabase.Password = "Metademo"

MyMetabase.Connect

'Define Query
Set MyQuery = MyMetabase.Queries.Add("Untitled1")
MyQuery.QueryCategories.Add "Brand"
MyQuery.QueryCategories.Add "Region"
MyQuery.QueryCategories.Item("Region").Orientation _
         = OrientationColumn
MyQuery.QueryItems.Add "Units Sold"

'Sort on Brand
MyQuery.QueryCategories.Item("Brand").SortDirection _
       = SortDirectionDesc
'Build Initial Report
Set MyMetaCube = MyQuery.MetaCubes.Add("Report1")
Worksheets.Item("Sort On Brand").Activate
Cells.Select
Selection.ClearContents

```

```
48  RunQuery MyMetaCube 'Calls a procedure for executing query
49
50  'Resort
51  MyMetaCube.SortColumnDirection = SortDirectionDesc
52  MyMetaCube.SortColumn = 1
53  'Rebuild Report
54  Worksheets.Item("Sort On Measure").Activate
55  Cells.Select
56  Selection.ClearContents
57
58  RunQuery MyMetaCube
59
60  'It Was No Fluke:  Resort on Brand
61  MyQuery.QueryCategories.Item("Brand").SortDirection _
62          = SortDirectionDesc
63  'Rebuild Report Again
64
65  Set MyMetaCube = MyQuery.MetaCubes.Add("Report2")
66  Worksheets.Item("Sort On Brand Again").Activate
67  Cells.Select
68  Selection.ClearContents
69
70  RunQuery MyMetaCube 'No effect: SortColumn overrides!
71
72  End Sub
73
74  Sub RunQuery(MyMetaCube)
75
76      'Declare Variables
77      Dim ReportRange As Range, MyData As Variant
78
79      'Transform Data in Cube into VB Array
80      Let MyData = MyMetaCube.ToVBArray
81
82      'Import Data into Excel Spreadsheet
83      Set ReportRange = _
84          ActiveSheet.Range _
85          (ActiveSheet.Cells(1, 1), _
86          ActiveSheet.Cells _
87          (MyMetaCube.Rows, MyMetaCube.Columns))
88      Let ReportRange.Value = MyData
89      ReportRange.EntireColumn.AutoFit  'Sizes columns
90
91   End Sub
```

Explanation of Exercise 23

Unlike previous exercises, this exercise involves a subroutine. The main procedure, **Sorting**, defines a query on **Brand** and **Region**, pivoting the **Region** attribute to the column orientation. The rows of the report are first sorted by **Brand**, using the QueryCategory object's **SortDirection** property, then sorted by measure, using the **SortColumn** property of the MetaCube object. Finally the sort on **Brand** is applied again to ascertain whether the order in which the commands are issued determines which property takes precedence.

Each sort generates a different MetaCube object, all of which are stored by the object variable **MyMetaCube** and processed by the subroutine **RunQuery**, which begins on line 74. Lines 48, 58, and 70 call this subroutine. When the **RunQuery** subroutine completes, the **Sort** procedure resumes execution at the point in the **Sort** procedure from which the call to that subroutine was made. Since all variables are declared locally, we must explicitly pass to the **RunQuery** subroutine any variables that the subroutine requires, in this case, the object variable **MyMetaCube**.

This exercise generates three reports. In the first report, generated by lines 39 through 44, the **SortDirection** property of the **Brand** QueryCategory sorts brands in a descending order:

Figure 8-19
Brand Sales by Region, Descending Sort on Brand Names (SortDirection Property of Brand QueryCategory

Region	Northeast	West
Brand	Units Sold	Units Sold
Techno Components	3699	5286
Suresound	2548	3464
Onetron	910	1254
NVD	2719	3788
Lasertech	1105	1665
Extreme	433	649

Region	Northeast	West
Delmore	1778	2557
Barton	1314	1924
Alden	1811	2626

As Figure 8-19 illustrates, a descending sort on the **Brand** attribute organizes the rows of the report such that brands appear in reverse-alphabetical order. Ordering the same rows by a different criterion, such as the numeric values within a particular column, abrogates the **SortDirection** property of the **Brand** QueryCategory instead of the **SortColumn** property of the MetaCube object. Although the **SortColumnDirection** property stores the same numeric constant, sorting brands according to how well they sold rather than the alphabetical order of their names obviously alters the arrangement of rows in the report.

In our case, line 52 identifies column one, or, since the count begins at zero, the second column from the left which is the column of numeric data on which to base the sort. As a result, the sort is based on brands' sales in the Northeast rather than the West region. Since both regions manifest the same trend in brand sales, this distinction is meaningless in this context.

Figure 8-20
Brand Sales by Region, Descending Sort on Brand Sales in Northeast (SortColumnProperty of MetaCube Object)

Region	Northeast	West
Brand	Units Sold	Units Sold
Techno Components	3699	5286
NVD	2719	3788
Suresound	2548	3464
Alden	1811	2626
Delmore	1778	2557

Region	Northeast	West
Barton	1314	1924
Lasertech	1105	1665
Onetron	910	1254
Extreme	433	649

As Figure 8-20 indicates, the **SortColumn** and **SortRow** properties of the MetaCube object override any conflicting sorts applied by the **SortDirection** property of the QueryCategory object.

MetaCube applies sorts on attribute values first and on numeric data second. This becomes important when performing sorts that are orthogonal to one another. Reversing the sort on a QueryCategory organized by columns changes the column specified by a **SortColumn** property; in the example, brands stored in rows are arranged according to how well they sold in the West rather than in the Northeast region.

To confirm that the order in which sort commands are issued is immaterial when two sorts are both applied to columns or both applied to rows, re-apply the sort to the **Brand** QueryCategory on lines 61 and 62. The report generated by this query is identical to its predecessor, shown as Figure 8-20.

In the unlikely event that a user specifies values for both the **SortColumn** property and the **SortRow** property of a single MetaCube object, MetaCube organizes columns first according to the relative values of a measure in a specified row. MetaCube then rearranges the order of the rows based on the relative values of a measure in a specified column. The **SortRow** property can thus determine the column on which the sorting of rows is based.

Whenever you perform numeric sorts on reports that organize multiple QueryCategory objects by either rows or columns, you must also use the **SortColumnBreaks** and **SortRowBreaks** properties to indicate whether multiple QueryCategories or only one QueryCategory should be affected by the numeric sort. Figure 8-21 on page 8-59 displays the result of a query executed with the **SortColumnBreak** property set to `False`.

Figure 8-21
Brand-Region Query: Sort on Numeric Column, SortColumnBreaks Set to False

Brand	Region	Units Sold
Extreme	Northeast	433
	West	649
Onetron	Northeast	910
Lasertech	Northeast	1105
Onetron	West	1254
Barton	Northeast	1314
Lasertech	West	1665
Delmore	Northeast	1778
Alden	Northeast	1811
Barton	West	1924
Suresound	Northeast	2548
Delmore	West	2557
Alden	West	2626
NVD	Northeast	2719
Suresound	West	3464
Techno Components	Northeast	3699
NVD	West	3788
Techno Components	West	5286

With the exception of the Extreme brand, whose sales were uniformly weak in both regions, the rows of the report in Figure 8-21 have been sorted without concern for displaying each brand's sales contiguously. Compare this to the report displayed in Figure 8-22. Here, the order in which brand names appear is unaffected by the sort.

Figure 8-22
Brand-Region Query: Sort on Numeric Column, SortColumnBreaks Set to True

Brand	Region	Units Sold
Alden	Northeast	1811
	West	2626
Barton	Northeast	1314
	West	1924
Delmore	Northeast	1778
	West	2557
Extreme	Northeast	433
	West	649
Lasertech	Northeast	1105
	West	1665
NVD	Northeast	2719
	West	3788
Onetron	Northeast	910
	West	1254
Suresound	Northeast	2548
	West	3464
Techno Components	Northeast	3699
	West	5286

The **SortRowBreaks** property serves the same purpose, determining how MetaCube performs numeric sorts on reports with multiple QueryCategory objects organized by columns.

MetaCubes Methods

The MetaCube object features a variety of methods to render data in different formats for different development environments. The MetaCube object's methods enable users to easily drill down or drill up to different levels of detail in a report. Figure 8-23 explains the methods of the **MetaCubes** class of objects.

Figure 8-23
MetaCubes Class of Objects: Methods

Method	Description/Example
AddDrill	Adds a new attribute value on which the analysis engine drills down or drills up, referring to the current **Row**, **Column**, and **Page** properties of the MetaCube object to identify that attribute value in the report. `MyMetaCube.AddDrill`
ClearDrills	Removes any previous row, page, or column references to cells on which the analysis engine has been set to drill down or to drill up. `MyMetaCube.ClearDrills`
ClearSorts	Abrogates any sort assigned by the **SortColumn** or **SortRow** properties of the **MetaCubes** class of objects. This method does not abrogate sorts assigned by the **SortDirection** property of the **QueryCategories** class of objects. `MyMetaCube.ClearSorts`
Copy	Copies the MetaCube object as a new instance of the same class, with the same properties, orientations, sorts, and summaries as the source object. The **Scratch** property of the new object is set to `True`. `Set MetaCopy = MyMetaCube.Copy`

(1 of 5)

Method	Description/Example

DrillDown	Returns a new object of the **MetaCubes** object class, implicitly defining a new query to retrieve detail-level data for a specified attribute value or set of attribute values. Referring to the **Row**, **Column**, and **Page** properties of the MetaCube object, the **AddDrill** method identifies each value on which the MetaCube analysis engine drills. This method requires two arguments, the item within the collection of **DrillDownAttributes** to which to drill from the selected attribute values and a Boolean flag indicating whether to include in the new report the values on which MetaCube drilled:
	`Set DetailCube = MyMetacube.DrillDown _`
	`(MyMetacube.DrillDownAttributes.Item(0), True)`
	See Exercise 24 on page 8-69 and the explanation that follows for a more thorough discussion of drilling down.

DrillUp	Returns a new object of the **MetaCubes** object class, implicitly defining a new query to retrieve summary-level data for a specified attribute value or set of attribute values. Referring to the **Row**, **Column**, and **Page** properties of the MetaCube object, the **AddDrill** method identifies each value on which the MetaCube analysis engine drills. This method requires one argument, the item within the collection of **DrillDownAttributes** to which to drill from the selected attribute values:
	`Set DetailCube = MyMetacube.DrillDown _`
	`(MyMetacube.DrillDownAttributes.Item(0), True)`
	See below for a more thorough discussion of drilling up.

Error-SpreadClip	This method returns as a tab-delimited string the error associated with a query result that has been extrapolated from a sample table. Query results extrapolated from sample tables include a range of error for each numeric value within the result. Attribute values and other non-numeric cells within the report return null values. For queries that do not process against a sample table, the **ErrorSpreadClip** method returns null values for all cells. For more information about sampling, query confidence, and error, see "The Samples Class of Objects" on page 6-30. The **Error-SpreadClip** method requires the same arguments as the more common **ToSpreadClip** method, described below.
	`Spread1.ClipValue = MyMetaCube.ErrorSpreadClip _`
	`(1, MyMetaCube.Rows)`

(2 of 5)

Method	Description/Example

ErrorVBArray	For each value in an extrapolated query result, this method returns a margin of error. The error values are assembled in a two- or three-dimensional variant array that can be stored by a Visual Basic 4.0 or Visual Basic for Applications variant or array variable. This method returns null values for queries that are not processed against a sample table. Non-numeric cells in a query result also return null error values, regardless of the table from which the result was retrieved or derived. For numeric query results that are extrapolated from a sample table, the error can be added or subtracted to that result, defining a range within which the actual value is likely to fall. The confidence with which MetaCube is required to certify that the actual result falls within that range determines the size of the range. For more information about sampling, query confidence, and error, see "The Samples Class of Objects" on page 6-30. The **ErrorVBArray** method can implicitly require MetaCube to execute a query.
	`Dim QueryData as Variant`
	`QueryData = MyMetaCube.ErrorVBArray`

FetchCell	Like the **Cell** property, this method returns, as a variant, the value of a specific cell within the virtual cube of data represented by a MetaCube object. You must specify as arguments the row, column and page number of the desired cell, in that order. Deploying this method before executing the query implicitly commands the MetaCube engine to execute the query. This method involves building the entire virtual cube on the client, and limits on the number of rows a query can retrieve still apply. `MyVariantValue = MyMetaCube.FetchCell (1, 1, 1)`

(3 of 5)

Method	Description/Example
FetchCellError	Like the **CellError** property, this method returns a number of the double type, indicating the error associated with a single value within an extrapolated query result. This method returns null values for queries that are not processed against a sample table. This method also returns null values for cells that are non-numeric, such as those that contain attribute values. For numeric query results that are extrapolated from a sample table, the error can be added or subtracted to the statistically predicted value, defining a range within which the actual value is likely to fall. The confidence with which MetaCube is required to certify that the actual result falls within that range determines the size of the range. For more information about sampling, query confidence, and error, see "The Samples Class of Objects" on page 6-30. This method, which can prompt MetaCube to execute a query, requires three arguments, specifying the row, column, and page number of the desired cell, in that order. `CellError = MyMetaCube.FetchCellError 1, 1, 1`
FetchCellType	Like the **CellType** property, this method returns an integer indicating the type of data a cell stores, whether it be a measure label, an attribute label, a measure, a value, and so on. Figure 8-18 on page 8-53 explains the significance of the integers returned by this method. This method requires the same arguments as the **FetchCell** method, and they must be specified in the same order. `CellCode = MyMetaCube.FetchCellType(1, 1, 1)`

(4 of 5)

Method	Description/Example

ToSpreadClip	Translates the data stored in the virtual cube represented by a MetaCube object into a tab-delimited string, which you can readily export to the popular Spread / VBX custom control. Deploying this method before executing the query implicitly commands the MetaCube engine to execute the query.
	`Spread1.Col = 1`
	`Spread1.Row = 1`
	`Spread1.Col2 = MyMetaCube.Columns`
	`Spread1.Row2 = MyMetaCube.Rows`
	`Spread1.ClipValue = MyMetaCube.ToSpreadClip _`
	`(1, MyMetaCube.Rows)`
	This example includes optional arguments. One specifies the row within the cube at which to begin exporting data, the other specifies the number of rows to export, where the default is all rows. Using these arguments to iteratively specify different chunks of data allows you to export large data sets to Spread / VBX.

ToVBArray	Translates the data represented by the MetaCube object as a virtual cube into a two- or three-dimensional variant array that can be stored in a Visual Basic 4.0 or Visual Basic for Applications array or variant variable. This method can implicitly require MetaCube to execute a query. For an example of such an application, see Exercise 3 on page 2-13.
	`Dim QueryData as Variant`
	`QueryData = MyMetaCube.ToVBArray`

(5 of 5)

The DrillDown Method

Drilling down enables you to navigate easily from a cell in a report to greater levels of detail, as displayed in a new report of the same format. The **DrillDown** method of the **MetaCubes** object class instantiates, defines and executes a new Query object identical to the parent of the MetaCube object except that one of the attributes in the original query is replaced by an attribute describing a lower level in the dimensional hierarchy.

Moreover, the new Query object includes a filter against the attribute on which you are drilling, retrieving only values within the range defined by the cells selected in the original report. If, in a monthly sales report, you drill down on the value July, the resulting report is grouped by the attribute describing the next level of detail within the time dimension, but only for the month of July—that is, only the days July 1 through July 31. Using the **AddDrill** method of the **MetaCubes** object class, you can specify more than one attribute value on which to drill, expanding the scope of the new, detail-level report.

The analytical engine generates a new MetaCube object to represent the data returned by the new query. The new MetaCube object features the same properties and describes the same report format as the previous MetaCube object, substituting the detail-level attribute for the summary-level attribute.

You can drill down on only values of a single attribute within the virtual cube represented by the MetaCube object. For each attribute value you must identify the cell containing such a value, defining its location in the three-dimensional virtual cube:

```
MyMetaCube.Column = 0
MyMetaCube.Row = 1
MyMetaCube.Page = 0
```

Once you have identified a cell storing an attribute value, you can deploy the **AddDrill** method:

```
MyMetaCube.AddDrill
```

This method requires no arguments, as the **Column**, **Row**, and **Page** properties of the MetaCube object already identify the cell on which to drill. For each value within an attribute on which you want to drill, you must specify a different cell using these properties, followed by the same **AddDrill** method. For a given MetaCube object, you cannot add drill directions for values of different attributes. Although you cannot view as a collection or ValueList the cells on which you are drilling, the values on which MetaCube drills are cumulative—the **AddDrill** method does not replace one value with the next. Once you have specified a cell on which to drill, you can exclude that cell from MetaCube's drill path only by clearing all drill paths using the **ClearDrills** method.

```
MyMetaCube.ClearDrills
```

Before deploying the **DrillDown** method, you must have identified at least one cell on which to drill, using the **Row**, **Column** and **Page** properties of the MetaCube object. The **DrillDown** method requires two arguments, the Attribute object to which you would like to drill, specified as an object, and a flag indicating whether existing attribute values should be included in the new report. This method returns a new MetaCube, named after the original MetaCube object, the parent of which is the new Query object. To execute the query, invoke the **ToVBArray** or **ToSpreadClip** method:

```
Set DetailLevelCube = MyMetaCube.DrillDown _
     (MyMetaCube.DrillDownAttributes.Item(0), True)
```

MyMetaCube's collection of DrillDownAttribute objects includes the default attributes to which you can drill for a specified cell. See Figure 8-24 on page 8-75. You can also specify the attribute on which to drill down by referring to the collection of attributes owned by a Dimension object.

DrillUp Method

Like the **DrillDown** method, this method returns a new instance of the MetaCube object with a new parent; both the Query object and the MetaCube object are identical to the original Query and MetaCube objects except that they represent data at a higher level of summarization.

Although the value or set of values from which you drill up in a report describes a particular level in the dimensional hierarchy—defining the collection of DrillUpAttributes—the resulting query is not filtered on that value. The new query thus retrieves data at a higher level of summarization but with the same scope. Since the subsequent query is not filtered on the selected attribute values, it is unnecessary to select more than one cell on which to drill.

As with the previous method, you must establish the cell from which you are drilling before deploying this method, using the **Row**, **Column**, and **Page** properties of the MetaCube object.

```
MyMetaCube.Row = 1
MyMetaCube.Column = 0
MyMetaCube.Page = 0
MyMetaCube.AddDrill
```

After you specify a cell on which to drill, deploy the **AddDrill** method. As with the **DrillDown** method, you cannot drill up from values of different attributes. The attribute values added to the drill path correspond to a particular level in the dimensional hierarchy, defining the collection of attributes reachable via the **DrillUp** method from that cell. We use this collection to specify the new Attribute object. The **DrillUp** method substitutes one of the attributes from this collection for the attribute from which you are drilling up:

```
Set DrillUpCube = MyMetaCube.DrillUp _
    (MyMetaCube.DrillUpAttributes.Item(0))
```

The **DrillUp** method returns a new MetaCube object named after the original MetaCube object, the parent of which is the new Query object. To execute the query, deploy the **ToVBArray** or **ToSpreadClip** method.

Exercise 24 illustrates many of the methods and properties used for drilling down and drilling up. The procedures in this exercise create a simple break report, with brands organized by rows and regions by columns, and then they drill down, first on brands and then on regions.

Exercise 24: Drilling Down

```
Option Explicit

'MetaCube API Exercise 24: Drilling Down

Sub MetaCube_API()

'Declare Variables and Constants
Dim MyMetabase As Object, MyQuery As Object, _
    MyMetaCube As Object
Const OrientationColumn = 2

'Login
Set MyMetabase = CreateObject("Metabase")

'Identify an ODBC Data Source
Let MyMetabase.ConnectString = "Metademo"

'Specify a set of metadata
Let MyMetabase.Name = "MetaCube Demo"

'Specify login to database
Let MyMetabase.Login = "metademo"

'Specify database password
Let MyMetabase.Password = "Metademo"

MyMetabase.Connect

'Define Query
Set MyQuery = MyMetabase.Queries.Add("Untitled1")
MyQuery.QueryCategories.Add "Brand"
MyQuery.QueryCategories.Add "Region"
MyQuery.QueryCategories.Item("Region").Orientation _
         = OrientationColumn
MyQuery.QueryItems.Add "Units Sold"

'Build Initial Report
Set MyMetaCube = MyQuery.MetaCubes.Add("Report1")
Worksheets.Item("Original Report").Activate
Cells.Select
Selection.ClearContents

'Call Procedure for Executing Query
RunQuery MyMetaCube


'Drill Down on Two Brands
```

```
'Add First Drill Value
    MyMetaCube.Row = 2 'First row has index number of zero
    MyMetaCube.Column = 0
    MyMetaCube.Page = 0
    MyMetaCube.AddDrill

'Add Second Drill Value
    MyMetaCube.Row = 3
    MyMetaCube.Column = 0
    MyMetaCube.Page = 0
    MyMetaCube.AddDrill

'Drill Away!
Set MyMetaCube = MyQuery.MetaCubes.Item("Report1") _
    .DrillDown(MyQuery.MetaCubes.Item("Report1") _
    .DrillDownAttributes.Item(0), _
    True)
Worksheets.Item("Drill Down").Activate 'New worksheet
Cells.Select
Selection.ClearContents

'Call Procedure for Executing Query
RunQuery MyMetaCube

'Drill Down on Region, From the Original Report

'Get Rid of Old Drills
MyQuery.MetaCubes.Item("Report1").ClearDrills

'Add Drill on a Region
    MyQuery.MetaCubes.Item("Report1").Row = 0
    MyQuery.MetaCubes.Item("Report1").Column = 1
    MyQuery.MetaCubes.Item("Report1").Page = 0
    MyQuery.MetaCubes.Item("Report1").AddDrill

'Drill Away!
Set MyMetaCube = MyQuery.MetaCubes.Item("Report1") _
  .DrillDown(MyQuery.MetaCubes.Item("Report1") _
  .DrillDownAttributes.Item(0), True)
Worksheets.Item("Drill Down Again").Activate
Cells.Select
Selection.ClearContents

'Call Procedure for Executing Query
RunQuery MyMetaCube

End Sub

Sub RunQuery(MyMetaCube)
```

```
97
98      'Declare Variables
99      Dim ReportRange As Range, MyData As Variant
100
101     'Transform Data in Cube into VB Array
102     Let MyData = MyMetaCube.ToVBArray
103
104     'Import Data into Excel Spreadsheet
105     Set ReportRange = _
106         ActiveSheet.Range _
107         (ActiveSheet.Cells(1, 1), _
108         ActiveSheet.Cells _
109         (MyMetaCube.Rows, MyMetaCube.Columns))
110     Let ReportRange.Value = MyData
111     ReportRange.EntireColumn.AutoFit  'Sizes columns
112
113 End Sub
```

Explanation of Exercise 24

Like the previous exercise, this exercise involves two separate procedures. The first procedure, **DrillDown**, defines a query and drills down on two different attributes, instantiating three MetaCube objects to store the result, all of which are stored in the object variable **MyMetaCube**. The **DrillDown** procedure is the main procedure and begins on line 8.

The second procedure, **RunQuery**, retrieves as an array the data represented by the three MetaCube objects defined in the main procedure, displaying that data in the currently active spreadsheet. This procedure begins on line 101. The **DrillDown** procedure calls the **RunQuery** procedure from lines 44, 70, and 92. Each call passes the object variable **MyMetaCube** to the **RunQuery** procedure. When the **RunQuery** procedure completes, the **DrillDown** procedure resumes execution at the point from which the call was made. As a subroutine to the main procedure, **RunQuery** cannot be executed independently.

The **DrillDown** procedure begins by declaring a set of object variables for storing Metabase, Query, and MetaCube objects. All variables are declared locally, requiring you to explicitly pass to the second procedure any variables that the subroutine requires. Line 10 declares a constant that you will later use to pivot the **Region** attribute to the column orientation.

Line 27 connects to the database, instantiating a multidimensional view of relational data in a Metabase object. As in previous exercises, no Metabase properties are set, and default connect parameters are read from the **metacube.ini** file. Lines 31 to 35 define the familiar **Brand-Region** query, and lines 38 and 39 instantiate a MetaCube object to represent the report. Although this MetaCube object differs from the objects that drilling down subsequently instantiates, the same object variable, **MyMetaCube**, iteratively stores each of the three objects.

Before it calls the **RunQuery** subroutine, the main procedure activates a worksheet, to which the subroutine returns data. With each call to the subroutine, a different worksheet is active, ensuring that no result overwrites its predecessor. Prior to running the main **DrillDown** procedure, you must have created three worksheets with the names **Original Report**, **Drill Down**, and **Drill Down Again**.

Line 44 calls the **RunQuery** subroutine to execute the query, passing the MetaCube object as an argument to the second procedure, which begins on line 96.

Because the calling application must pass the object variable **MyMetaCube** to the **RunQuery** subroutine, the **MyMetaCube** object variable appears in the parentheses following the name of the procedure.

On line 102, the **RunQuery** procedure declares two variables: one to store data from **MyMetaCube**, the other to define a range of cells in which that data is displayed. Line 102 prompts MetaCube to retrieve data from the database as the **ToVBArray** method returns data as an array to the **MyData** variant variable. Lines 105 to 111 define a report range based on the number of rows and columns in the query result and assign each value in the array to a different cell in the spreadsheet. At the end of this procedure on line 111, the **DrillDown** procedure resumes execution on line 47.

Using the **Row**, **Column**, and **Page** properties, lines 49 to 51 define a position within the original MetaCube object's multidimensional structure that stores an attribute value, in this case the Alden brand. Upon specifying a cell, line 52 invokes the **AddDrill** method, thereby including that attribute value in the group of attribute values on which the **DrillDown** method will act. In a similar fashion, lines 54 to 58 include a second attribute value, Barton, in this group. The **DrillDown** method on line 62 thus retrieves detailed information for two brands, Alden and Barton.

Once the cells on which to drill are set, you can deploy the **DrillDown** method. Drilling down instantiates a new MetaCube object with a new parent, a Query object with characteristics identical to its predecessor's, except that a new filter has been applied to include data only for the values being drilled down on, and a new, lower-level QueryCategory has been included to show data in more detail.

For efficiency, the **MyMetaCube** object variable stores the new MetaCube object, replacing the original MetaCube object. Although the application itself obviates the original MetaCube object, it is preserved in memory by the analysis engine as an item within a collection. Immediately you must refer to this object in line 63, as the original MetaCube object's collection of **DrillDown** attributes determines which attributes are available for inclusion in the drill down query.

This collection of **DrillDown** attributes consists of the default attributes for each dimension element one level below the level of the attribute on which you drilled. In this case, there is only one dimension element directly below the level of the **Brand** attribute, and that dimension element's default attribute is **Product**. We thus specify the first and only item within this collection as our drill down attribute. The second argument in this statement, a Boolean flag set to `True`, directs MetaCube to include the attributes on which we are drilling in the new drill down report.

After activating a new worksheet, the procedure again calls the **RunQuery** subroutine, retrieving data for the MetaCube object as before.

After executing the new query, the **DrillDown** procedure resumes on line 75, using the **ClearDrills** method to eliminate the attribute values of the previous query from the drill path. The original MetaCube must be identified as an item within a collection because the object variable **MyMetaCube** now stores the MetaCube object generated by the first drill down.

Otherwise, this section of code, which is included only to demonstrate that the original MetaCube object remains in memory, is identical to the previous section; a position is defined within the multidimensional result set of the original MetaCube object, a drill direction is added, and then MetaCube drills down from that position to a specified attribute, again preserving in the new report the value on which the analysis engine drills.

MetaCubes Collections

The objects within the MetaCube object's collections allow you to perform different calculations and manipulations on a given set of data without requerying the database. Also included as denormalized, read-only collections of the MetaCube object are collections describing the attributes to which you can drill up or drill down from different cells within the virtual cube of data represented by the MetaCube object. The latter collections can really be thought of as belonging to different cells within a cube, as their content changes from cell to cell.

Figure 8-24 summarizes the MetaCubes object class collections.

Figure 8-24
MetaCubes Class of Objects: Collections

Collection	Description/Example
DrillDown-Attributes	Consists of a subset of Attribute objects to which you can drill down from a given cell. The composition of the collection depends on the cell specified by the **Row**, **Column**, and **Page** properties of the MetaCube object. The collection thus changes from cell to cell, as identifying a cell containing a value of an attribute different than the previous cell changes the collection of attributes reachable via drill down from that cell. You cannot directly add or delete Attribute objects to this collection.
DrillUp-Attributes	Consists of a subset of Attribute objects to which you can drill up from a given cell. The composition of the collection depends on the cell specified by the **Row**, **Column**, and **Page** properties of the MetaCube object. The collection thus changes from cell to cell, as identifying a cell containing a value of an attribute different than the previous cell changes the collection of attributes reachable via drill up from that cell. You cannot directly add or delete Attribute objects to this collection.

Summaries	Consists of objects that subtotal break reports. For example, a report subdividing brand sales by region could perform a subtotal on the QueryCategory object representing brand, calculating each brand's total or average sales for all regions and storing the result in a row interpolated at each brand value. Summary objects can also perform grand totals on columns or rows, as well as minimums, maximums, counts, and averages. When instantiating a Summary object, you must include as arguments the QueryCategory object and a constant indicating the type of calculation to perform on each grouping within the report.
	`MyMetaCube.Summaries.Add MyQueryCategory, _`
	`SummaryTotal`

The Summaries Class of Objects

For MetaCube objects that represent break reports, the Summary object performs calculations on the attribute values within a larger grouping, summing, averaging, or counting their associated records. For weekly sales, subdivided by state, you can calculate the average state sales for each week, the total state sales for each, or the number of states for which sales are recorded each week. To perform any of these calculations, instantiate a Summary object by specifying the QueryCategory object representing the **Week** attribute as the attribute for which you want to perform the calculations. A second argument indicates the type of calculation to perform on the values within each **Week** grouping:

```
MyMetaCube.Summaries.Add MyQueryCategory, SummaryTotal
```

The Summary object can also calculate grand totals, averages, counts, minimums and maximums for all columns or all rows, in which case the first argument is null, irrespective of the groupings within a break report. Should you want to perform several different calculations on the same QueryCategory object, you can simply instantiate additional Summary objects, specifying a different type of calculation for each. Each of the arguments in the instantiation command correspond to a Summary object property, as explained in Figure 8-25

Figure 8-25
Summaries Class of Objects: Properties

Properties	Description/Example
QueryCategory	Object. Identifies the QueryCategory object representing the attribute for which the calculation is to be performed. All subrows within the grouping defined by values of the attribute will be summed, averaged, or counted. `Set MySummary.QueryCategory = MyQueryCategory`
Parent	Returns the MetaCube object. `MsgBox MyMetaCube.Parent.Name`
SummaryType	Integer. Indicates the type of calculation to perform, as correlated to the numeric values identified in Figure 8-26. `MsgBox MyMetaCube.SummaryType = SummaryCount`

Figure 8-26 summarizes the constants for the **SummaryType** property.

Figure 8-26
Summaries Class of Objects: SummaryType Constants

Type of Summarization	MetaCons.bas Constant Name	Constant
Subtotal	SummaryTotal	1
Average	SummaryAverage	2
Count	SummaryCount	3
Minimum	SummaryMin	4
Maximum	SummaryMax	5
Grand Total, All Rows	SummaryRowGrandTotal	11
Average, All Rows	SummaryRowGrandAverage	12
Count, All Rows	SummaryRowGrandCount	13
Minimum Value, Among All Rows	SummaryRowGrandMin	14
Maximum Value, Among All Rows	SummaryRowGrandMax	15
Grand Total, All Columns	SummaryColumnGrandTotal	21
Average, All Columns	SummaryColumnGrand-Average	22
Count, All Columns	SummaryColumnGrandCount	23
Minimum Value, Among All Columns	SummaryColumnGrandMin	24
Maximum Value, Among All Columns	SummaryColumnGrandMax	25

Summary objects that calculate subtotals and averages require you to specify the name of the attribute for which to calculate the average or subtotal. But, as Figure 8-26 indicates, the Summary object can perform many calculations that do not require a QueryCategory object as an argument. In place of a QueryCategory object, Visual Basic developers can simply write `Nothing` or some other value that the development environment's editor accepts.

The Summary object does not feature any methods, nor does it own any collections.

The QueryBackJobs Class of Objects

Submitting a query for background processing instantiates a QueryBackJob object with the same name as the saved query. The QueryBackJob object does not feature an add method, as each job is submitted and retrieved by deploying Query object methods.

The Metabase object organizes its QueryBackJob collection by user, only showing those QueryBack jobs submitted by a particular user. You cannot create public QueryBackJobs. The Query object organizes its QueryBackJob collection by query, such that each collection only consists of the QueryBack jobs spawned from that Query object.

QueryBackJobs Properties

The properties of a QueryBackJob object describe the status of a query submitted to QueryBack. After a user submits a query to QueryBack, a server-side scheduling daemon that periodically identifies any new jobs assigns the query a unique job identification number. If the user schedules the job to run immediately, the scheduling daemon places the job on a queue of jobs submitted by other users, arranging jobs on the queue according to their priority. When a server processor becomes available, the scheduling daemon spawns a process to execute the query, storing the result on the database. If a user schedules a job to run in the future, the scheduling daemon stores the query's definition until the time specified, at which time the job is placed on the queue as before.

Because all of the properties of a QueryBackJob object depend on the status of server-side processes, you cannot assign values to these properties and they are, in effect, read-only. To retrieve values for a QueryBackJob object's properties, you must first deploy the **RefreshStatus** method, as explained below.

Figure 8-27 summarizes the properties of the **QueryBackJobs** class of objects:

Figure 8-27
QueryBackJobs Class of Objects: Properties

Property	Description/Example
ErrorCode	Integer. Stores any error codes returned by the server while processing a QueryBack job. `MsgBox MyQueryBackJob.ErrorCode`
ErrorText	String. Stores any error message returned by the server while processing a QueryBack job. `MsgBox MyQueryBackJob.ErrorText`
JobID	Long integer. The unique identification number assigned to each QueryBack job by the scheduling daemon upon submission. `MsgBox MyQueryBackJob.JobID`
Name	String. Indicates the name of the QueryBack job, which corresponds to the original name of the Query object submitted for background processing. The default property. `MsgBox MyQueryBackJob.Name`
Parent	Object. Metabase object. `MsgBox MyQueryBackJob.Parent.Name`

Priority	Integer. Indicates the priority assigned to the job upon submission.
	`If MyQueryBackJob.Priority > 3 _`
	`Then MsgBox "Query will run soon."`

RecurType	Integer. Indicates the frequency with which the scheduler re-executes the query, as signified by the numeric values explained in Figure 8-3 on page 8-20. Each recurrence of a QueryBack job essentially automatically instantiates a new job, with a new identification number, and so on. Defaults to no recurrence.
	`If MyQueryBackJob.RecurType = RecurTypeNone _`
	`Then MsgBox "The query not scheduled to run again."`

Size	Long. Read-only. Stores the number of rows that a QueryBackJob object returns to the client. If the QueryBack job has not executed on the server, the value of this property is `0`. `MsgBox MyQueryBackJob.Size`

(1 of 2)

Property	Description/Example
StartTime	Date variant. Indicates the time at which the QueryBack job actually began processing on the server. Null, pending execution. `MsgBox MyQueryBackJob.StartTime`
Status	Integer. Indicates whether a QueryBack job is pending, executing, or complete, as signified by one of the numeric constants described in Figure 8-28 on page 8-81. Jobs that have been submitted to run at a later date, as well as jobs on the scheduler's queue, are identified as pending. `MsgBox MyQueryBackJob.Status`
StopTime	Date variant. Indicates the time at which the server finished processing the query. `MsgBox MyQueryBackJob.StopTime`
SubmitTime	Date variant. Indicates the time at which the user submitted the job. `MsgBox MyQueryBackJob.SubmitTime`
TargetStart	Date variant. Indicates the time at which the user requested that the QueryBack job begin processing, as specified in one of the **Submit** method's arguments. Differences between the value of this property and that of the **StartTime** property depend on the length of the job queue. `MsgBox MyQuery.TargetStart`

(2 of 2)

Related Numeric Constants

Figure 8-28 on page 8-81 explains the significance of the three numeric values possible for the QueryBackJob object's **Status** property, listing the constant names for these values as they are found in the **MetaCons.bas** file in your MetaCube directory.

Figure 8-28
QueryBackJobs Class of Objects: Status Constants

Job Status	MetaCons.bas Constant Name	Constant
Pending	QueryBackJobStatusPending	0
Executing	QueryBackJobStatusRunning	1
Completed	QueryBackJobStatusFinished	2

QueryBackJobs Methods

Figure 8-29 summarizes the methods of the **QueryBackJobs** class of objects.

Figure 8-29
QueryBackJobs Class of Objects: Methods

Method	Description/Example
DeleteJob	Deletes a pending or completed job from the queue. Deleting a recurring job prevents that job from spawning a new incarnation of itself for the following, day, month, or year. When applied to a completed QueryBack job, this method deletes whatever tables store their results. `MyQueryBackJob.DeleteJob`
RefreshStatus	Retrieves and refreshes values of the QueryBackJob object's properties. Before performing any operation on an existing QueryBack job, you must deploy this method. You can also deploy this method on an entire collection of QueryBackJob objects. `MyMetabase.QueryBackJob.RefreshStatus`
Retrieve	This method retrieves from the database the result of a QueryBack job, returning a Query object and, if the **Scratch** property was set to `False` prior to submission, any children of that object that existed when the query was submitted. `Set MyQuery = MyQueryBackJob.Retrieve` For an example of an application that submits a query to QueryBack and retrieves the result, see "Exercise 21: Submitting a Query to QueryBack" on page 8-17.

QueryBackJobs Collections

Figure 8-30 summarizes the methods available for **QueryBackJobs** collections:

Figure 8-30
QueryBackJobs Class of Objects: Collections

Collection	Description
RefreshStatus	Retrieves and refreshes the values returned by all properties for all QueryBackJob objects in a collection. `MyMetabase.QueryBackJobs.RefreshStatus`
ItemJobID	Object. the unique identification number that the scheduling daemon assigns to each QueryBackJob object when the job is submitted. This number is the index into the QueryBackJobs collection. `MyMetabase.QueryBackJobs.ItemJobID`

Chapter

9

The Schemas Class of Objects and Its Collections

In This Chapter

This chapter introduces the **Schemas** class of objects and its hierarchy of **Tables** and **Columns** collections. All three object classes feature a **Name** property by which you can view the names of the schemas / table owners, tables, and columns in the relational database. MetaCube Warehouse Manager invokes these objects to populate the Physical Object Map, a view of schemas / table owners, tables, and columns in the database.

This entire class of objects and its descendants are provided for your reference when creating the metadata map of the physical structures within the relational database. MetaCube creates the collections of Schema, Table, and Column objects by reviewing the database system tables. You thus cannot instantiate an object within any of these collections except indirectly, by creating a new table owner, a table, or a column in the relational database. You also cannot change the properties of these objects; they are read-only.

Schemas, Tables, Columns

The **Schemas** class of objects features only a single property, **Name**, which stores the string name of a schema / table owner in the relational database. Within the collection of Schema objects, there is a Schema object for each schema / table owner in the database's system tables. Each Schema object owns a collection of Table objects identifying the tables within that schema / table owner.

Like the **Schemas** object class, the **Tables** class of objects features only a single property, the **Name** property, which identifies the name of the table that the object represents. Each table so identified stores a set of columns, described by the Table object's collection of Column objects.

The Column object features two properties: the **Name** property, which identifies the name of a column in the table owning that Column object, and the **Type** property, which indicates the type of data stored in the column. The **Type** property identifies different column types by an integer code, summarized in Figure 9-1.

Figure 9-1
Column Type Constants

Column Type	MetaCons.bas Constant Name	Constant
Character or Variable Character	DataTypeCharacter	0
Numeric	DataTypeNumeric	1
Date	DataTypeDate	2
Other	DataTypeUnsupported	3

Chapter

10

Users and DSS Systems

In This Chapter

This chapter introduces the **DSSSystems** class of objects, the **Users** class of objects, and the **AvailableDSSSystems** class of objects, a child of the **Users** class.

In MetaCube Secure Warehouse, an administrator manages access to data by assigning DSS Systems to users or users to DSS Systems. The administrator can further restrict the availability of data by assigning mandatory filters to a user of a DSS System. By defining a users's properties in Secure Warehouse, the administrator specifies how that user can interact with an Informix database. All of these Secure Warehouse procedures are accomplished by manipulating the **Users**, **DSSSystems** and **AvailableDSSSystems** classes of objects.

The DSSSystems Class of Objects

The **DSSSystems** class of objects is used in MetaCube Secure Warehouse to represent DSS Systems that are available in metadata for a particular database connection. In Secure Warehouse, the DSS Systems listed in the DSS Systems tree correspond to the member objects of a **DSSSystems** collection.

You cannot deploy an **Add** method to instantiate a DSSSystem object. Instead, after MetaCube connects to the database, the **DSSSystems** collection is populated with DSSSystem objects defined in metadata. To create a new DSSSystem object, use **Metabase.CreateNew**, as described in "Metabase Methods" on page 3-11.

There are few manipulations you can perform on a DSSSystem object; the class has only two properties and no methods.

DSSSystem Properties

Figure 10-1 describes the two properties available for the **DSSSystems** class of objects.

Figure 10-1
DSSSystems Class of Objects: Properties

Property	Description/Example
LastUpdate	Stores, as a variant, the date and time the DSS System's metadata was last updated. `MsgBox MyDSSSystem.LastUpdate`
Name	A string that identifies the name of the DSS System. `MyDSSystem.Name = "Finance Demo"`

DSSSystems Collections

In Secure Warehouse, the **DSSSystems** collection represents all of the DSS Systems available for a particular database connection. Although the **DSSSystems** collection of objects has no **Add** method, it does provide all the other standard methods available for manipulating MetaCube collections, such as **MakeFirst**, **MakeNth**, and **Remove** (see Figure 1-1 on page 1-7). It also includes the standard properties **Count** and **Names** (see "Object Class Hierarchies and Collections" on page 1-6).

The Users Class of Objects

The **Users** class of objects supports the actions a data warehouse administrator would perform while using Secure Warehouse, such as assigning DSS Systems and mandatory filters to users and defining user properties. In Secure Warehouse, the users listed in the Users tree correspond to the member objects of a **Users** collection. To manipulate users, you must be a secure user—that is, the data warehouse administrator must use MetaCube Secure Warehouse to grant you access to Secure Warehouse and Warehouse Manager.

Instantiating a User Object

To instantiate a User object, add a new instance of the **Users** class of objects to a Metabase object's collection of users:

```
MyMetabase.Users.Add "User1"
```

When instantiating a user, you must include the name of the user as an argument. To call the **Add** method, a user must be granted access to Secure Warehouse (that is, the property **User.SecureUser** must be set to `True`).

Users Properties

A User object combines three categories of information: DSS Systems available to this user (represented by the **AvailableDSSSystems** collection, which is a child of the User object), mandatory filters (assigned using methods available in the **Users** class), and user properties.

The properties of a User object control how a user interacts with an Informix database. For example, one property defines the PDQ priority to use while processing a user's queries. Figure 10-2 summarizes the properties of the **Users** class of objects.

Figure 10-2
User Class of Objects: Properties

Property	Description/Example
AuditUser	Boolean. A `True` value indicates that MetaCube records information about queries run by the user. This information can be used to tune system performance. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.AuditUser = True`
ConnectString	String. Identifies the ODBC data source; defaults to the value of **Metabase.ConnectString**. `MyUser.ConnectString = "MetaDemo"`
DataSkip	Long. Determines whether an Informix RDBMS can skip locked or otherwise unavailable rows when attempting to retrieve data for the user. Figure 3-5 on page 3-15 identifies the possible constants for this property. DataSkip is enabled or disabled when the user connects to MetaCube and an Informix RDBMS. Defaults to the server setting. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.DataSkip = DataSkipOff` For more information, see the DATASKIP entry in the *Informix Guide to SQL: Syntax*.
DefaultDSS	String. Identifies the name of the default DSS System for the user. The default value for this property is `Null`. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.DefaultDSS = "Finances"`

(1 of 5)

Property	Description/Example
ForeignUser	Boolean. A `True` value for this read-only property indicates the current user exists in the client table for the current database connection and an entry for the current user exists in the MetaCube registry, but the **ConnectString** information for the user, as stored in the registry, does not match the connect string used for the current database connection. `MyUser.ForeignUser = True`
MandatoryQB	Boolean. A `True` value indicates that this user cannot run queries in real time and must generate QueryBack jobs. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.MandatoryQB = True`
MaxTotal-Fetches	Long. A number greater than zero that determines the maximum number of rows that MetaCube retrieves for a single SQL statement. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.MaxTotalFetches = 200`
MetaSchema	String. The prefix applied to MetaCube metadata tables to ensure their uniqueness in the RDBMS; defaults to the value of **Metabase.MetaSchema**. `MyUser.MetaSchema = "MetaCube."`
Name	String. The name of the User object, specified as an argument upon instantiation. This name should be unique within a **Users** collection, and its uniqueness will be enforced when you use the **User.Save** method. `MyUser.Name = "New Name"`

(2 of 5)

Property	Description/Example
PDQPriority	Long. An integer from -1 to 100. This property designates the parallel database query (PDQ) priority of the users's decision support system queries submitted by MetaCube to the Informix database. PDQ Priorities determine the extent to which the Informix database executes a user's queries in parallel. A value of 0 explicitly precludes any parallel operations, a value of 1 enables only parallel scans, and values between 2 and 100 represent the percent of available system resources that queries against this decision support system can consume. In multiprocessor systems, high PDQ priorities enable the database to process queries faster. The value of this property defaults to -1, indicating that MetaCube does not set PDQ priority for a user. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to True) can call this method. `MyUser.PDQPriority = 50` For more information about PDQPriority, see the *Administrator's Guide for Informix Dynamic Server*.
QBPriority	Long. The priority to be assigned to QueryBack jobs submitted by the user. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to True) can call this method. `MyUser.QBPriority = 3`
QBSpace	String. The dbspace to be used for objects created for the user. An empty string indicates that the default dbspace should be used. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to True) can call this method. `MyUser.QBSpace = ""`

(3 of 5)

Property	Description/Example
QueryBack-Times	String. Identifies the hours when this user's QueryBack jobs can execute. The string consists of triplets, that is, multiples of three numbers: ■ The first number specifies the day of the week. Possibilities range from `1` to `7`, where `1` = Monday and `7`= Sunday. ■ The second number specifies the start time, as measured in seconds (`0` to `86,399`) ■ The third number specifies the stop time, also measured in seconds (`0` to `86,399`) The numbers within each triplet are delimited by slashes. The triplets themselves are delimited by backslashes. For example, `1/0/86399\2/0/86399` consists of two triplets that allow the user to execute QueryBack jobs all day on Monday and Tuesday. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.QueryBackTimes = "1/0/86399\2/0/86399"`
QueryBack-TimesCount	A read-only property provided to help client applications parse the **QueryBackTimes** string. `MgsBox MyUser.QueryBackTimesCount`
Role	String. The name of the Informix role assigned to the user. An empty string indicates that no role is assigned. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.Role = ""`

(4 of 5)

Property	Description/Example
SecureUser	Boolean. A true value indicates this user is authorized to access Secure Warehouse and MetaCube Warehouse Manager. The default value is `False`. During upgrades from previous MetaCube versions, the user **metapub** is always set to `True`. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.SecureUser = True`
SlowQuery-Warning	Long. An integer greater than zero that defines the threshold for issuing a slow query warning. MetaCube Explorer displays a slow query warning when a user submits a query with a processing cost greater than this threshold value. Processing costs are based on values that the data warehouse administrator assigns to metadata tables. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.SlowQueryWarning = 10000`
UserSetPDQ	Boolean. A `True` value indicates that the user can set his or her own PDQ priority in a client application. Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method. `MyUser.UserSetPDQ = True`

(5 of 5)

Users Methods

Figure 10-3 summarizes the methods of the **Users** class of objects.

Figure 10-3
User Class of Objects: Methods

Method	Description/Example

AddMandatory-Filter	Adds a mandatory filter to the collection of filters owned by the user. The method requires two arguments: a string providing the full path to the filter and a string providing the owner of the filter. The owner should always be **metapub**. The full path to a filter is obtained by deploying **Filter.FullPathName**.
	Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method.
	`MyUser.AddMandatoryFilter _`
	`"\Mandatory\Time\LastYear", "metapub"`

Delete	Deletes a user from the RDBMS metadata and deletes that user's entry from the MetaCube engine's registry. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to `True`) can call this method. `MyUser.Delete`
Mandatory-FilterNames	Retrieves a ValueList of the names and paths of mandatory filters. The method takes one argument: the DSSSystem object for which filters should be retrieved. `MsgBox MyUser.MandatoryFilterNames MyDSSSystem`

(1 of 2)

Method	Description/Example
Mandatory-FilterOwners	Retrieves a ValueList of the owners of mandatory filters. The method takes one argument: the DSSSystem object for which filters should be retrieved. Each item in the ValueList corresponds to an item in the ValueList generated by **MandatoryFilterNames**. The order of both lists is identical and cannot be changed. `MsgBox MyUser.MandatoryFilterOwners MyDSSSystem`

Remove-MandatoryFilter	Removes a mandatory filter from the collection of mandatory filters owned by this user. This method *does not* remove this filter from the RDBMS metadata. The method requires two arguments: a string providing the full path to the filter and a string providing the owner of the filter. The full path to a filter is obtained by deploying **Filter.FullPathName**.
	Only users who have been granted access to Secure Warehouse (that is, the property **User.SecureUser** is set to `True`) can call this method.
	`MyUser.RemoveMandatoryFilter _`
	`"\Mandatory\Time\LastYear", "metapub"`

Save	Saves the user to the RDBMS metadata. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to `True`) can call this method. `MyUser.Save`

(2 of 2)

Users Collections

In Secure Warehouse, the **Users** collection represents all of the users with entries in the MetaCube registry. Typically, this corresponds to all the users who are being managed with Secure Warehouse. The **Users** collection provides all the standard methods available for manipulating MetaCube collections, such as **MakeFirst**, **MakeNth**, and **Remove** (see Figure 1-1 on page 1-7), and it also includes the standard properties **Count** and **Names** (see "Object Class Hierarchies and Collections" on page 1-6). In addition to the standard methods and properties available to all collections, the **Users** collection provides the methods summarized in Figure 10-4.

To call the **Remove** method, a user must be granted access to Secure Warehouse (that is, the property **User.SecureUser** must be set to `True`).

Figure 10-4
Users Class of Objects: Methods

Method	Description/Example
LoadUsers	Loads all user information contained in the metadata client table; typically used for Secure Warehouse only. No standard MetaCube applications except Secure Warehouse need the user information stored in the client table, so to improve performance, the MetaCube analysis engine normally does not load that information into memory. Custom-built applications, however, might require the user information available in the client table. All user information stored in the registry of the MetaCube analysis engine is always available, and that information is sufficient to display a list of users in Secure Warehouse. To access user properties, however, the **LoadUsers** method must be deployed. `MyUsers.LoadUsers`
PurgeRegistry	Removes user entries from the MetaCube registry. This method is used to clean up the registry when a DSS System or database connection is no longer used. Which user entries are actually removed by this method depends on this method's one argument, a string that contains the connect string for a database connection. Based on that string, MetaCube does either of the following: ■ If the connect string does not match the connect string for the current connection, MetaCube purges from its registry all user entries containing the specified connect string. ■ If the connect string does match the connect string for the current connection, MetaCube purges from its registry all entries for users who do not have an entry in the **Client** table but who do have a connect string that matches the specified connect string. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to `True`) can call this method. `MyUsers.PurgeRegistry "MetaCube Demo"`

Method	Description/Example
SynchRegistry	Adds users to the MetaCube registry in bulk. When called, this method adds entries to the registry for all users who exist in the Client table for the current database connection but do not have an entry in the registry. Once a user has a registry entry, that user is displayed in MetaCube Secure Warehouse and can connect to a database using Web Explorer. Only users who have been granted access to Secure Warehouse (meaning the property **User.SecureUser** is set to `True`) can call this method. `MyUsers.SynchRegistry`

The AvailableDSSSystems Class of Objects

Each **AvailableDSSSystems** class of objects is a child of a User object. An AvailableDSSSystems object is a collection representing the DSS Systems that a particular user can access. The members of an **AvailableDSSSystems** collection are actually pointers to objects in the **DSSSystems** collection that is owned by the Metabase object. (For more information, see "The DSSSystems Class of Objects" on page 10-3.) In Secure Warehouse, the **AvailableDSSSystems** collection represents all of the DSS Systems that have been assigned to a user, thus granting that user access to those DSS Systems.

Instantiating an AvailableDSSSystem Object

To instantiate an AvailableDSSSystem object, add an instance of the **AvailableDSSSystems** class of objects to a User object:

```
MyMetabase.User.AvailableDSSSystems.Add MyDSSSystems
```

When instantiating an AvailableDSSSystem object, you must provide a DSSSystem object as an argument.

AvailableDSSSystems Properties and Methods

The **AvailableDSSSystems** collection provides all the standard methods available for manipulating MetaCube collections, such as **MakeFirst**, **MakeNth**, and **Remove** (see Figure 1-1 on page 1-7), and it also includes the standard properties **Count** and **Names** (see "Object Class Hierarchies and Collections" on page 1-6).

Chapter

11

The SystemMessages Class of Objects

In This Chapter

This chapter introduces the **SystemMessages** class of objects, which allows you to distribute messages to all users within a DSS System. MetaCube Agent Administrator, MetaCube's client-side tool for database administrators, enables administrators to create messages that users can view through MetaCube Explorer's interface. Both applications either get or set properties of the SystemMessage object.

The SystemMessages Class of Objects

To distribute a new message throughout the MetaCube family of applications, you must instantiate a new SystemMessage object. To instantiate a SystemMessage object, you must enter the message itself as the first argument and the date and time as the second. The **CurrentTime** property of the Metabase object can furnish the latter argument:

```
MyMetabase.SystemMessages.Add _
    "The Data Warehouse is operational!", _
        MyMetabase.CurrentTime
```

You need not specify the name of the SystemMessage object since this class of objects does not feature a **Name** property. If you find it necessary to identify a particular SystemMessage object, do so by index number. Aside from the standard **Parent** property, the string text of the message and its date are the only properties of the SystemMessage object, as explained in Figure 11-1 on page 11-4.

Figure 11-1
SystemMessages Class of Objects: Properties

Property	Description/Example
LastUpdate	Date variant: The time and date of the message. `MySystemMessage = MyMetabase.CurrentTime`
Message	String: The text of the message. Default property. `MySystemMessage.Message = "Up and running"`
Parent	The Metabase object. The scope of a message is limited to a DSS System. `MsgBox MySystemMessage.Parent.Name`

The SystemMessage object features no methods and owns no collections.

Chapter

12

The ValueList

In This Chapter

This chapter introduces the ValueList.

The ValueList

To return multiple values into different development environments, MetaCube allows you to specify the format in which to return those values. Any property which stores a set of values as a ValueList can include an additional term in its syntax indicating that the values should be stored as an array or a tab-delimited string. In addition, you can specify a subset of the values in the ValueList, the maximum number of values to return, or the order in which to return them.

The objects that return a ValueList and the properties of those objects are:

- **Dimension.AttributeNames**
- **Attribute.ValueList**
- **FactTable.MeasureNames**
- **MetaCube.FormatStrings**
- **Extension.Arguments**
- **Extension.ArgumentTypes**
- **Extension.Functions**
- **Extension.Types**
- **MetaCube.PageLabels**
- All objects' **VerifyResults** properties

Figure 12-1 summarizes the commands that can be appended to a statement returning a ValueList. Three of the five commands apply to only the actual ValueList property of the Attribute object. The remaining commands, or specifications, are illustrated with an example involving an instantiation of the **Dimensions** class of objects, as represented by an object variable named **MyDimension**, and that object's **AttributeNames** property. You can substitute any of the objects and their respective properties listed on the previous page.

Figure 12-1
ValueList Specifications

Specification	Description/Example

ArrayValues	This specification translates the values represented in a ValueList to an array, which can be stored in a variant or array variable in Visual Basic for Applications and in other variable types in other development environments. Visual Basic 3.0 does not support the passing of arrays through object linking and embedding. You can translate the values in any ValueList to an array.
	`Let VariantVariable = _`
	`MyDimension.AttributeNames.ArrayValues`

MaxRows	Treat this command as a property of the Attribute object's **ValueList** property, which represents all the values for the attribute. **MaxRows** can be set equal to any long value. **MaxRows** limits the number of values that the **ValueList** property can return, a function useful for preventing tab-delimited strings from becoming unwieldy. The default is `200`. You cannot set a maximum on the values in ValueLists returned by properties other than Attribute object's **ValueList** property.
	`Let MyAttribute.ValueList.MaxRows = 2`
	`MsgBox MyAttribute.ValueList.TabbedValues`

(1 of 2)

Specification	Description/Example

Sort	This command specifies a sort on the values represented in the Attribute object's **ValueList** property. Set the **Sort** command equal to a long value or corresponding constant, as specified in Figure 8-7 on page 8-25. You cannot sort values in ValueLists represented by any property other than the **ValueList** property of the Attribute object.
	`Let MyAttribute.ValueList.Sort = SortDirectionDesc`
	`MsgBox MyAttribute.ValueList.TabbedValues`

Subset	This command allows you to use SQL wildcard syntax to specify a subset of the values in an Attribute object's ValueList. You can apply this command only to a ValueList represented by an Attribute object's **ValueList** property.
	`Let MyAttribute.ValueList.Subset = "%x%"`
	`MsgBox MyAttribute.ValueList.TabbedValues`
	This example only displays values that include the letter "x."

TabbedValues	This specification translates the values represented by a ValueList into a tab-delimited string. If you do not indicate how MetaCube should return the values in a ValueList, MetaCube, by default, returns the values as a tab-delimited string. You can deploy this command on any ValueList, as returned by any of the properties listed above.
	`Let StringVariable = _`
	`MyDimension.AttributeNames.TabbedValues`
	or
	`Let StringVariable = MyDimension.AttributeNames`

(2 of 2)

Chapter

13

The Applications Class of Objects and Global Properties

In This Chapter

This chapter briefly explains the Applications class of objects, the highest-level object class for any OLE software server. For all MetaCube applications this object represents the MetaCube analysis engine. We previously introduced the **Metabase** class of objects as the parent of all other object classes in MetaCube's hierarchy. For the purposes of application development this is always the case. However, the OLE standard requires the **Applications** object class, which we include here for the sake of completeness.

This chapter also introduces two properties that apply to all MetaCube object classes.

The Applications Class of Objects

Figure 13-1 summarizes the properties of the **Applications** class of objects.

Figure 13-1
Applications Class of Objects: Properties

Property	Description/Example
FullName	String. This property returns the file name and location of the MetaCube engine, including the path. Typically, the value of this read-only property is `c:\metacube\metacube.exe`. `MsgBox Application.FullName`
Name	String. This read-only property returns the name of the application, in this case, `MetaCube`. This property is the default property of the Application object. `MsgBox Application.Name`
Parent	Object. This property returns the parent of the Application object, which is the Application object itself. This property is required by OLE but is otherwise useless.
Visible	Boolean. Returns `False`. Required by OLE.

The **Applications** class of objects features no methods and only one collection, the collection of Metabase objects currently active. When deploying properties of the **Applications** object class in Visual Basic for Applications, you will notice that the name of the application returned is Excel, as Excel and MetaCube communicate continuously. In an application compiled as an executable, the Application object represents the MetaCube analysis engine installed in your computer.

Global Properties: Application and Type

All MetaCube object classes have in common two properties, the **Application** property and the **Type** property. The **Application** property returns the name of the application, `MetaCube`. The **Type** property returns the name of the object class of which the object is an instance, such as `Measures` or `DimensionElements`. Both properties are read-only and return strings.

Chapter

14

Scoping Rules

In This Chapter

This chapter briefly explains rules for identifying objects that have the same name but belong to a different parent or belong to a different class entirely.

Scoping Rules

To avoid ambiguity when specifying the name of a DimensionElement, Attribute, or Measure object, follow the conventions discussed below.

The DimensionElements Object Class

You can identify a DimensionElement object by name if the name of the DimensionElement object is unique throughout the DSS System. If dimension elements belonging to different dimensions are described by objects of the same name, you can specify the name of the parent Dimension object. If a DimensionElement object and an Attribute object belonging to the same dimension share the same name, you can also indicate that you refer to the DimensionElement object, not the Attribute object. To identify a dimension element named **Brand** within the **Product** dimension, three names are possible, listed in order of increasing scope and precision:

```
MyQuery.QueryCategories.Add "Brand"
MyQuery.QueryCategories.Add "Product.Brand"
MyQuery.QueryCategories.Add _
    "DimensionElements.Product.Brand"
```

The Attributes Object Class

You can identify an Attribute object by name if the name of the Attribute object is unique throughout the DSS System. If attributes belonging to different dimensions are described by objects of the same name, you can specify the name of the parent Dimension object. If an Attribute object and a DimensionElement object belonging to the same dimension share the same name, you can also indicate that you refer to the Attribute object, not the DimensionElement object. To identify an attribute named **Brand** within the **Product** dimension, three names are possible, listed in order of increasing scope and precision:

```
MyQuery.QueryCategories.Add "Brand"
MyQuery.QueryCategories.Add "Product.Brand"
MyQuery.QueryCategories.Add _
    "Attributes.Product.Brand"
```

The Measures Object Class

You can identify a Measure object by name if the name of the Measure object is unique throughout the DSS System. If measures belonging to different fact tables are described by objects of the same name, you can specify the name of the parent FactTable object. If a Measure object and some other object associated with the fact table share the same name, you can also indicate that you refer to an object of the **Measures** class.

To identify a measure named **Units Sold** within the **Sales Transaction** fact table, three names are possible, listed in order of increasing scope and precision:

```
MyQuery.QueryItems.Add "Units Sold"
MyQuery.QueryItems.Add _
    "Sales Transactions.Units Sold"
MyQuery.QueryItems.Add _
    "Measures.Sales Transactions.Units Sold"
```

It is good programming practice to scope as precisely as possible. For simplicity's sake, some examples in this text have not been precisely scoped.

Index

E

F

G

I

L

M

O

P

Q

R

S

T

U

V

X