12

T-SQL Programming Objects

SQL Server is an enormously capable and powerful relational data store. In general, SQL Server manages transactions and enforces checks and rules to protect the integrity of related records and values. You've seen how the query optimizer makes intelligent decisions and uses indexes to make queries run fast and efficiently. Now we're going to take SQL Server to the next level. Most data is accessed through business applications. SQL Server can be more than just an idle medium for storing this data. A well-designed business solution uses the capabilities of an active database server, programming objects, and other components to distribute the workload and minimize unnecessary network traffic.

I want to take you on a brief tour of history so that you can appreciate the impact of the features we're about to discuss. In the 1980s and early 1990s, PC-based applications ran only on desktops. If data could be shared across networks, it was simply stored in files managed by the file system. Applications supported a small number of users and quickly choked low-bandwidth networks as they moved all of their data to each desktop for processing. Desktop database applications sprang up like weeds in a new garden as inexpensive business applications became available — but the industry quickly hit the technology wall. In the past decade, the PC platform came of age with the advent of client/server database systems. In a nutshell, the enabling technology behind client/server applications was the cutting-edge concept of running application code on a database server. Products such as SQL Server enabled this capability through the use of database programming objects such as views and stored procedures.

I could stop there and keep things quite simple, but the current state of the industry has moved forward in recent years. Most enterprise database solutions have progressed beyond simple client/server technology. Now it's easier than ever before to distribute program components across two, three, or more different computers. These may include desktop computers, Web servers, application servers, and database servers.

Sophisticated database applications use complicated queries. For this reason, it is important that queries and other SQL logic are protected and run as efficiently as possible. If SQL statements are managed in server-side database objects rather than in applications, this reduces the overall complexity of a solution. This separation of client-side applications and databases enables programmers and database professionals to each do what they do best, rather than having to write both program code and complex SQL — not to mention the fact that application programmers, unless they have a background in database technologies, have traditionally written very bad SQL.

The very first rule of developing database applications is to avoid the ad-hoc query at all costs. Ad-hoc queries create great efficiency issues, and when it comes to Web applications, great security issues as well. The best practice when creating database-centric applications is to use database programming objects. In SQL Server, these objects include views, stored procedures, functions, and triggers.
This chapter covers each of these objects in turn.

Views

A view is one of the simplest database objects. On the surface, a view is nothing more than a `SELECT` query that is saved with a name in a database. Ask any modern-day programmer what they believe to be the most important and fundamental concept of programming. They will likely tell you that it is code reuse. Writing every line of code, every object, every script, and every query represents a cost or risk. One risk is that there could be a mistake (a bug) in the code. The cost of a bug is that it must be fixed (debugged) and tested. Buggy applications must be redeployed, shipped, installed, and supported. Undiscovered bugs pose a risk to productivity, business viability, and perhaps even legal exposure. One of the few constants in the software universe is change. Business rules will change, program logic will change, and the structure of your databases will also change. For all of these and other reasons, it just makes sense to reduce the number of objects that you create and use in your solutions. If you can create one object and reuse it in several places rather than duplicating the same effort, this limits your exposure to risk. Views promote this concept of code reuse by enabling you to save common queries into a uniform object. Rather than rewriting queries, complex queries can be created and tested and then reused without the added risk of starting over the next time you need to add functionality to an application.

Virtual Tables

One of the great challenges facing users is dealing with the complexity of large business databases. Many tools are available for use by casual database consumers for browsing data and building reports. Applications such as Microsoft Excel and Access are often used by information workers, rather than programmers, to obtain critical business management and operational information. A typical mid-scale database can contain scores of tables that contain supporting or special-purpose data. To reassemble the information stored in a large database, several tables must be joined in queries that take even skilled database professionals time and effort to create effectively. As you've seen in many examples, this is often not a trivial task. From the user's perspective, views are tables. They show up in most applications connecting to a SQL Server, along with the tables. A view is addressed in a `SELECT` statement and exposed columns, just like a table.

From the developer or database designer's perspective, a view can be a complex query that is exposed as if it were a simple table. This gives you an enormous amount of flexibility and the ability to hide all of the query logic, exposing a simple object. Users simply see a table-like object from which they can select data.

Creating a View

Defining a view is quite simple. First of all, a database user must be granted permission to create a view. This is a task that you may want to have performed only by a database administrator or a select number of trusted users. Because creating a view isn't particularly complicated, you may want certain users to be granted this ability.

Several simplified tools are available that you can use to create views. Microsoft Access, SQL Server Management Studio, and Visual Studio all leverage the T-SQL Query Designer interface to create and manage views. The process is just about the same in all of these tools because they all actually expose the same components. The following section steps through creating a view using Management Studio. I will not demonstrate each tool because the process is nearly identical.

Creating a View in Management Studio

Creating a view with Management Studio is very easy using the graphical query designer. It is basically the same designer used by Visual Studio and Microsoft Access. In Management Studio's Object Explorer, navigate to the AdventureWorks2008 database and then expand the database to expose the Views folder. Right-click the Views folder and choose New View, as shown in Figure 12-1.

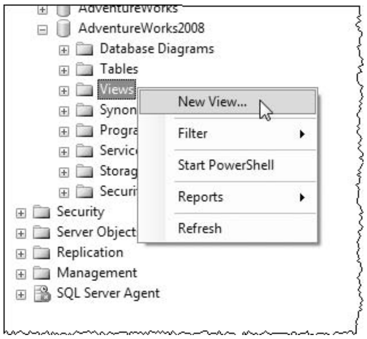


Figure 12-1

After clicking New View, the Add Table dialog appears, as shown in Figure 12-2. Add the Product, ProductCategory, and ProductSubcategory tables. Then click the Close button.

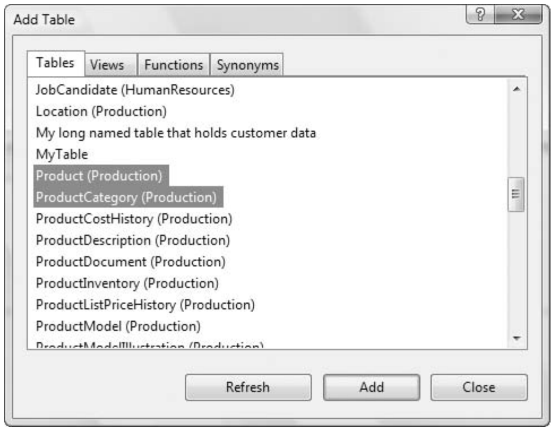
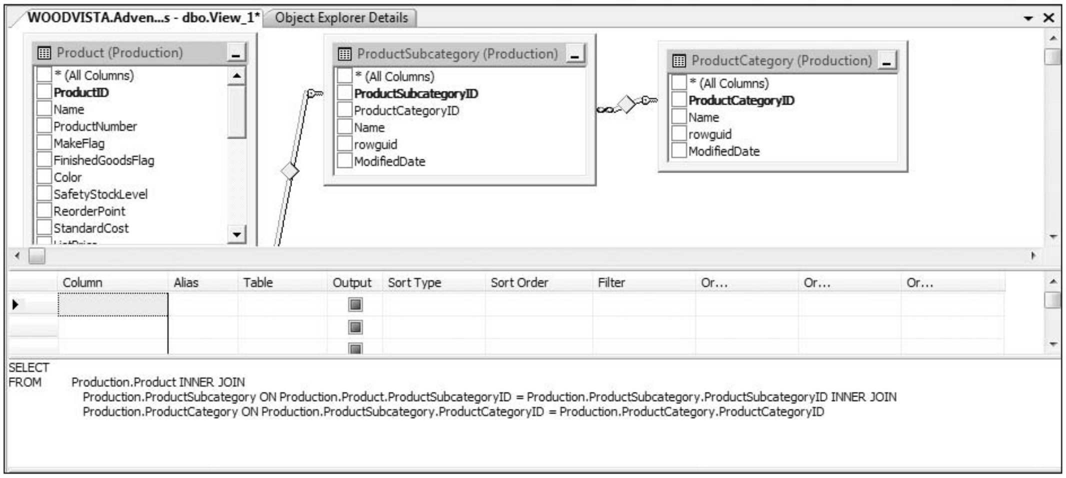

Add Table
Tables
Views
Functions
Synonyms
JobCandidate (HumanResources)
Location (Production)
My long named table that holds customer data
MyTable
Product (Production)
ProductCategory (Production)
ProductCostHistory (Production)
ProductDescription (Production)
ProductDocument (Production)
ProductInventory (Production)
ProductListPriceHistory (Production)
ProductModel (Production)
Refresh
Add
Close

Figure 12-2

After you close the Add Table dialog, the graphical query designer is displayed. It shows the three tables connected by relation links in the top Diagram pane. Because of the relationships that exist between these tables, inner joins are automatically defined in the query. Beneath the Diagram pane is the Criteria pane, SQL pane, and then the Show Results pane, as shown in Figure 12-3.



```
SELECT
FROM       Production.Product INNER JOIN
             Production.ProductSubcategory ON Production.Product.ProductSubcategoryID = Production.ProductSubcategory.ProductSubcategoryID INNER JOIN
             Production.ProductCategory ON Production.ProductSubcategory.ProductCategoryID = Production.ProductCategory.ProductCategoryID
```

Figure 12-3

Each one of the panes can be displayed or hidden by clicking on the associated button on the View Designer toolbar (see Figure 12-4).

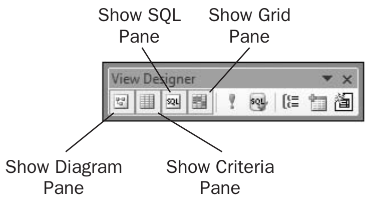
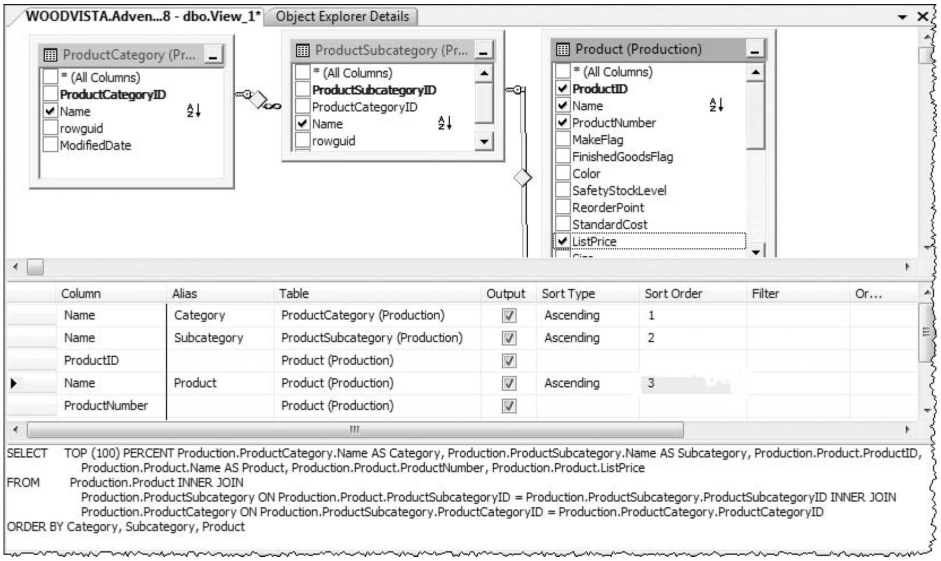


Figure 12-4

Select the Name column from the ProductCategory table and the Name column from the ProductSubcategory table. Then select the ProductID, Name, ProductNumber and ListPrice columns from the Product table (using the checkboxes in the table windows). Using the Alias column in the columns grid, define aliases for the following three columns:

Table.Column	Alias
ProductCategory.Name	Category
ProductSubcategory.Name	Subcategory
Product.Name	Product

Also, designate these three columns for sorting in the order listed by dropping down and selecting the word Ascending in the Sort Type column. Check your results against Figure 12-5 and make any adjustments necessary.



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
Name	Category	ProductCategory (Production)	✓	Ascending	1		
Name	Subcategory	ProductSubcategory (Production)	✓	Ascending	2		
ProductID		Product (Production)	✓				
Name	Product	Product (Production)	✓	Ascending	3		
ProductNumber		Product (Production)	✓				

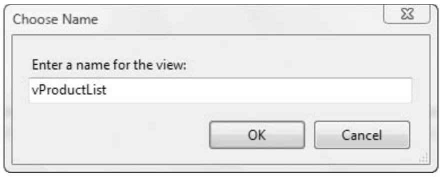
```
SELECT    TOP (100) PERCENT Production.ProductCategory.Name AS Category, Production.ProductSubcategory.Name AS Subcategory, Production.Product.ProductID,
              Production.Product.Name AS Product, Production.Product.ProductNumber, Production.Product.ListPrice
FROM       Production.Product INNER JOIN
              Production.ProductSubcategory ON Production.Product.ProductSubcategoryID = Production.ProductSubcategory.ProductSubcategoryID INNER JOIN
              Production.ProductCategory ON Production.ProductSubcategory.ProductCategoryID = Production.ProductCategory.ProductCategoryID
ORDER BY Category, Subcategory, Product
```

Figure 12-5

Notice that something interesting happens to the SQL that is being written for you when you choose sort criteria: the query designer adds the `TOP 100 PERCENT` statement to your code.

Back when the original ANSI SQL specification was written, its authors wanted to ensure that database designers wouldn't create SQL queries that would waste server resources. Keep in mind that this was at a time when production servers had 32MB of memory. One common memory-intensive operation is reordering a large result set. So, in their infinite wisdom, the authors imposed a rule that views cannot support the `ORDER BY` clause unless the results are restricted using a `TOP` statement.

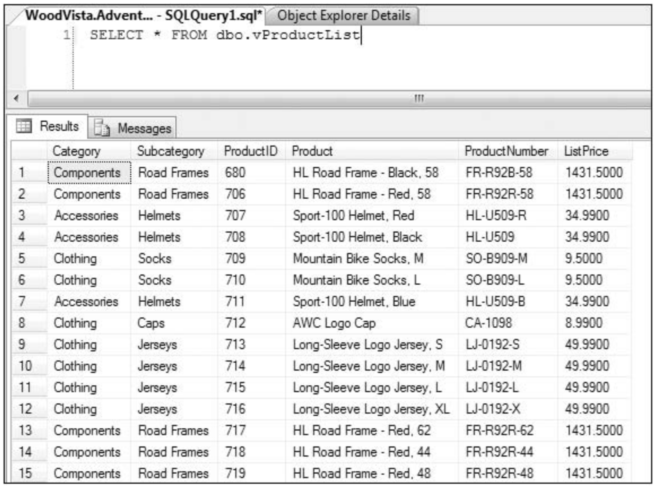
If you close the window, using the Close button in the top-right corner, Management Studio prompts you to save the view. Click Yes to save the view and enter a name for the new view in the Choose Name dialog, as shown in Figure 12-6. I've always made it a point to prefix view names with v, vw, or vw_ and to use Pascal case (no spaces, with the first letter of each word capitalized). This ensures that when you retrieve objects with older data access methods, the drivers rarely differentiated between tables and views. When creating a data application, it is generally pretty important to know if the object you're referencing is a physical table or a view.



Enter a name for the view: vProductList

Figure 12-6

After you save the view, it appears in the list of views in Object Explorer and can now be queried as you would query a table, as shown in Figure 12-7.



```
SELECT * FROM dbo.vProductList
```

	Category	Subcategory	ProductID	Product	ProductNumber	ListPrice
1	Components	Road Frames	680	HL Road Frame - Black, 58	FR-R92B-58	1431.5000
2	Components	Road Frames	706	HL Road Frame - Red, 58	FR-R92R-58	1431.5000
3	Accessories	Helmets	707	Sport-100 Helmet, Red	HL-U509-R	34.9900
4	Accessories	Helmets	708	Sport-100 Helmet, Black	HL-U509	34.9900
5	Clothing	Socks	709	Mountain Bike Socks, M	SO-B909-M	9.5000
6	Clothing	Socks	710	Mountain Bike Socks, L	SO-B909-L	9.5000
7	Accessories	Helmets	711	Sport-100 Helmet, Blue	HL-U509-B	34.9900
8	Clothing	Caps	712	AWC Logo Cap	CA-1098	8.9900
9	Clothing	Jerseys	713	Long-Sleeve Logo Jersey, S	LJ-0192-S	49.9900
10	Clothing	Jerseys	714	Long-Sleeve Logo Jersey, M	LJ-0192-M	49.9900
11	Clothing	Jerseys	715	Long-Sleeve Logo Jersey, L	LJ-0192-L	49.9900
12	Clothing	Jerseys	716	Long-Sleeve Logo Jersey, XL	LJ-0192-X	49.9900
13	Components	Road Frames	717	HL Road Frame - Red, 62	FR-R92R-62	1431.5000
14	Components	Road Frames	718	HL Road Frame - Red, 44	FR-R92R-44	1431.5000
15	Components	Road Frames	719	HL Road Frame - Red, 48	FR-R92R-48	1431.5000

Figure 12-7

Creating a View Using SQL Script

Regardless of the tool or product used to create a view, as you saw in the previous example, SQL script runs in the background and the result will be as varied as handwriting without the use of an automated tool. The syntax for creating a new view is quite simple. The pattern is the same whether the query is very simple or extremely complex. I'll start with a simple view on a single table:

```
CREATE VIEW vProductCosts
AS
SELECT ProductID, Name, StandardCost
FROM Production.Product
```

To continue working with this view and extend its capabilities, I can either use the `ALTER` command to make modifications to the existing view or drop and create it. Using the `ALTER` statement rather than dropping and re-creating a view has the advantage of keeping any existing properties and security permissions intact.

Here are examples of these two statements. The `ALTER` statement is issued with the revised view definition:

```
ALTER VIEW vProductCosts
AS
SELECT ProductID, ProductSubcategoryID, Name, ProductNumber, StandardCost
FROM Production.Product
```

Using the `DROP` statement will wipe the slate clean, so to speak, reinitializing properties and security permissions. But for the sake of the following example, don't run this code just yet.

```
DROP VIEW vProductCosts
```

What happens if there are dependencies on a view? I'll conduct a simple experiment by creating another view that selects data from the view previously created:

```
CREATE VIEW vProductCosts2
AS
SELECT Name, StandardCost FROM vProductCosts
```

For this view to work the first view has to exist and it must support the columns it references. Now, what happens if I try to drop the first view? I'll execute the previous `DROP` command. Here's what SQL Server returns:

```
Command(s) completed successfully.
```

The view is gone? What happens if I execute a query using the second view?

```
SELECT * FROM vProductCosts2
```

SQL Server returns this information:

```
Msg 208, Level 16, State 1, Procedure vProductCosts2, Line 3
Invalid object name 'vProductCosts'
Msg 4413, Level 16, State 1, Line 1
Could not use view or function 'vProductCosts2' because of binding errors.
```

Why would SQL Server allow something so silly? I may not be able to answer this question to your satisfaction because I can't answer the question to my own satisfaction. This capability to drop an object and break something else is actually documented as a feature call ***delayed resolution***. It's a holdover from the early days of SQL Server, but to a degree it makes sense. The perk of this feature is that if you needed to write script to drop all of the objects in the database and then create them again, this would be difficult to pull off with a lot of complex dependencies. If you're uncomfortable with this explanation, there is good news. An optional directive on the `CREATE VIEW` statement called `SCHEMA BINDING` tells SQL Server to check for dependencies and disallow any modifications that would violate them. To demonstrate, the first thing I'll do is drop both of these views and then re-create them:

```
CREATE VIEW vProductCosts WITH SCHEMABINDING
AS
SELECT ProductID, ProductSubcategoryID, Name, ProductNumber, StandardCost
FROM Production.Product
GO
CREATE VIEW vProductCosts2 WITH SCHEMABINDING
AS
SELECT Name, StandardCost
FROM dbo.vProductCosts
```

Some unique requirements are apparent in the example script. First of all, for a view to be schema-bound, any objects it depends on must also be schema-bound. Tables inherently support schema binding, but views must be explicitly schema-bound.

Any dependent objects must exist in the database before they can be referenced. For this reason, it's necessary to use batch delineation statements between dependent `CREATE` object statements. This example used the `GO` statement to finalize creating the first view.

When referring to a dependent view, you must use a two-part name. This means that you must use the schema name (which, as you can see in this example, is dbo). A schema-bound view also cannot use the `SELECT *` syntax. All columns must be explicitly referenced.

Ordering Rows

As mentioned earlier when working with the View designer, ordering rows in a view is not allowed without the `TOP` statement.

I run into this restriction all of the time. I'll spend some time creating some big, multi-table join or subquery with ordered results. After it's working, I think, "Hey, I ought to make this into a view." So I slap a `CREATE VIEW vMyBigGnarlyQuery AS` statement on the front of the script and execute the script with this result:

```
The ORDER BY clause is invalid in views, inline functions, derived tables,
subqueries, and common table expressions, unless TOP or FOR XML is also
specified.
```

Then I remember I have to use a `TOP` statement. This is a no-brainer and is easily rectified using the following workaround:

```
CREATE VIEW vOrderedProductCosts
AS
SELECT TOP 100 PERCENT ProductID, Name, ProductNumber, StandardCost
FROM Production.Product
ORDER BY Name
```

Now that most database servers have 500 times the horsepower and 100 times the memory of those 10 to 15 years ago, ordering a large result set is of much lesser concern.

Partitioned Views

Every system has its limits. Performance-tuning and capacity planning is the science of identifying these gaps and formulating appropriate plans to alleviate them. To partition data is to place tables or other objects in different files and on different disk drives to improve performance and organize data storage on a server. One of the most common methods to increase the performance and fault-tolerance of a database server is to implement RAID storage devices. Although this isn't a book on server configuration, I bring this up for a good reason. In teaching classes on database design and talking about partitioning data across multiple hard disks, I've often heard experienced students ask, "Why don't you just use a RAID device? Doesn't it accomplish the same thing?" Yes, to a point. Disk arrays using RAID 5 or RAID 10 simply spread data across an array of physical disks, improving performance and

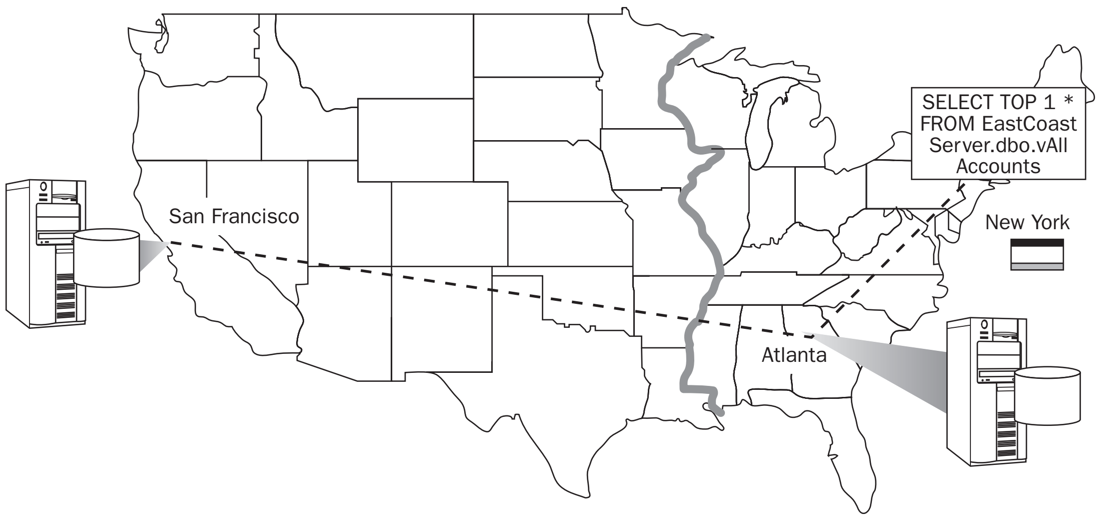


Figure 12-14

Securing Data

Another useful offering of views is to provide a layer for user data access without giving users access to sensitive data or other database objects. A common security practice is for the database administrator to lock down all of the tables, denying access to all regular users. Views are then created to explicitly expose selected tables, columns, and/or rows for all or selected users. When the select permission is granted on a view, users gain access to the view's underlying data even if the same user is explicitly denied the select permission on the underlying table(s).

Hiding Complexity

One of the most common, and arguably one of the most important, reasons to use views is to simplify data access. In a normalized database, even the most basic queries can involve many different tables. Views make it possible for programmers, report writers, and users to gain access to data at a reasonably low level without having to contend with the complexities of relationships and database schema.
A practical transactional database is broken down into many tables and related information is spread out across these tables to maintain data integrity and to reduce unnecessary redundancy. Reassembling all of these elements can be a headache for someone who doesn't fully understand the data or who may not be versed in relational database design. Even for the experienced developer or DBA, using a view can save time and minimize errors. The following is an example to demonstrate this point. To do something as fundamental as return product inventory information can be a relatively complex proposition.

In this example, I want to return the category, subcategory, product, model, shelf location, inventory quantity, inventory cost, and inventory date for all products. Because this information could be common to more than one product, the category, subcategory, location cost, and inventory data is stored in separate tables. For operational reasons, I have decided to exclude the price information as well as other descriptive data for the products. Using a view, users are not even aware that the columns containing the omitted information exist.

```
CREATE VIEW vProductInventory
AS
SELECT PC.Name AS Category
      ,SC.Name AS Subcategory
      ,P.Name AS Product
      ,P.ProductNumber
      ,PM.Name AS Model
      ,PN.Shelf
      ,PN.Quantity
      ,P.StandardCost
      ,P.StandardCost * PN.Quantity AS InventoryValue
      ,MAX(PN.ModifiedDate) AS InventoryDate
FROM   Production.Product P
INNER JOIN Production.ProductInventory PN
  ON P.ProductID = PN.ProductID
INNER JOIN Production.ProductModel PM
  ON P.ProductModelID = PM.ProductModelID
INNER JOIN Production.ProductSubcategory SC
  ON P.ProductSubcategoryID = SC.ProductSubcategoryID
INNER JOIN Production.ProductCategory PC
  ON SC.ProductCategoryID = PC.ProductCategoryID
GROUP BY PC.Name
        ,SC.Name
        ,P.Name
        ,P.ProductNumber
        ,PM.Name
        ,PN.Shelf
        ,PN.Quantity
        ,P.StandardCost
        ,P.StandardCost * PN.Quantity
```

Here's one more example of a lengthy view. This view will be used a little later in the discussion on processing business logic in stored procedures. I like this view because it contains several columns that can easily be used for reporting purposes, and to sort, group, or filter the resulting data.

```
CREATE VIEW vProductSalesDetail
AS
SELECT DISTINCT
       PC.Name AS Category
      ,PS.Name AS Subcategory
      ,P.Name AS Product
      ,ST.Name AS Territory
      ,SUM(SOD.OrderQty) AS TotalQuantity
      ,SUM(SOD.UnitPrice) AS TotalPrice
      ,SUM(SOD.UnitPriceDiscount) AS TotalDiscount
      ,SUM(SOD.LineTotal) AS LineTotals
      ,SOH.OrderDate
      ,SUM(SOH.SubTotal) AS SubTotal
      ,SUM(SOH.TaxAmt) AS TotalTax
      ,SUM(SOH.TotalDue) AS TotalDue
FROM   Production.ProductCategory PC
```

```
   INNER JOIN Production.ProductSubcategory PS
     ON PC.ProductCategoryID = PS.ProductCategoryID
   INNER JOIN Production.Product P
     ON PS.ProductSubcategoryID = P.ProductSubcategoryID
   INNER JOIN Sales.SalesOrderDetail SOD
     ON P.ProductID = SOD.ProductID
   INNER JOIN Sales.SalesOrderHeader SOH
     ON SOD.SalesOrderID = SOH.SalesOrderID
   INNER JOIN Sales.SalesTerritory ST
     ON ST.TerritoryID = SOH.TerritoryID
   GROUP BY PC.Name
           ,PS.Name
           ,P.Name
           ,ST.Name
           ,SOH.OrderDate
```

Modifying Data Through Views

Can data be modified through a view? Perhaps a better question is *should* data be modified through a view? The definitive answer is maybe. Yes, you can modify some data through views. Because a view can expose the results of a variety of query techniques, some results may be updatable, some may not, and others may allow some columns to be updated. This all depends on various join types, record-locking conditions, and permissions on the underlying tables.

As a rule, I don't think views are for updating records — that's my opinion. After all, doesn't the word view suggest that its purpose is to provide a read-only view of data? I think so, but I've also worked on enough corporate production databases where this was the only option.

The fact of the matter is that over time, databases evolve. Over the years, people come and go, policies are implemented with little evidence of their purpose, and political culture dictates the methods we use. If I ruled the world, no one would have access to data directly through tables; views would provide read-only data access and support all related application features, and stored procedures would be used to perform all transactional operations and filtered data retrieval. These are the guidelines I follow when designing a system from the ground up. However, I acknowledge that this is not always possible in the typical circumstance where one database designer isn't given free license.

In simple terms, these are the most common rules governing the conditions for updating data through views:

- ❑ In an inner join, columns from one table at a time may be modified. This is due to the record-locking restrictions on related tables. Updates generally cannot be performed on two related tables within the same transaction.
- ❑ In an outer join, generally columns only for the inner table are updatable.
- ❑ Updates can't be performed through a view containing a `UNION` query.