**Lesson 1: Designing Views**

Estimated lesson time: 40 minutes

Views are often called *virtual tables* because they can be referenced inside the T-SQL language whenever a table name is expected. This means that SQL Server 2005 enables you to encapsulate (or package) a specific T-SQL query inside a view and reference it anywhere else in the database without having to declare the T-SQL query again. For database developers, there are several scenarios in which this is valuable:

* Hiding implementation details as a matter of security
* Code maintainability
* Code reusability
* Backward compatibility when maintaining multiple versions of client applications not aware of changes in the database schema

In this lesson, you will look at the key design decisions to consider, depending on the scenario, when designing a view to retrieve data in SQL Server 2005.

**Choosing Between the Different Types of Views**

T-SQL offers the CREATE VIEW statement to define any type of view. Depending on certain implementation and deployment differences, views can be standard views, indexed views, or partitioned views. These implementation details are also used by the SQL Server query optimizer to decide the best strategy for executing the code contained in the view.

When designing views, you must have a clear understanding of the view’s purpose, the scenario that it fulfills, where the data is located, and the security context under which it will execute. These details enable you to choose carefully the type of view that will provide the most benefit according to the requirements of the scenario that you are designing for.

**Designing Standard Views**

You use a standard view when you want to package a T-SQL query as a unit for security, deployment, and reusability. A standard view is the most common type of view, fitting most scenarios.

Standard views store only the encapsulated T-SQL query code instead of storing the resulting data. When referenced from another query, the view is expanded in place with the actual T-SQL query code it contains so that the SQL Server 2005 query processor compiles a single execution plan for the whole query. Because standard views are always materialized at run time, the execution cost is exactly the same as if the actual T-SQL query had been fed to the query engine.

The main benefits of encapsulating a query in a view are:

* Control over what you want to make visible outside of the database to protect the inner complexities of the database schema (this includes data structures as well as naming conventions). Acting as an abstraction layer, a view provides a public interface to the outside world, and you, as the designer, decide what is exposed and how.
* By hiding the database schema, the view protects outside data consumers and client applications from schema changes and gives database designers the ability to make changes to inner details without affecting consumers.

The following code example uses a standard view to filter the sales data by showing only the sales orders coming from the Northeast territory:

CREATE VIEW [Sales].[vNortheastSalesOrderHeader]

AS

SELECT

FROM Sales.SalesOrderHeader

WHERE (TerritoryID IN

(SELECT TerritoryID

FROM Sales.SalesTerritory

WHERE (Name LIKE N'Northeast')))

**On the Companion Disc**

This chapter includes many code examples. You will find all the code from this chapter on the companion CD in the C:\My Documents\Microsoft Press\TK70-441 \Chapter06\Sql folder.

Because standard views do not occupy disk storage space, you can define as many views as needed without affecting system performance, so always design views that return just the right amount of data.

**IMPORTANT Maximum number of objects**

The sum of the number of all objects in a database cannot exceed 2,147,483,647.

When designing a view, consider that it can provide different ways of looking at the data. You can use a view to pre-combine some values, pre-aggregate data, or consolidate multiple tables to provide a new perspective on the data.

A view represents an external interface that protects outsiders from schema changes and from understanding the inner complexity of the physical schema or abstracts such changes and complexity. This is an important concept and technology because it uncouples applications from the table’s schema, which is harder to maintain and modify.

**Designing Indexed Views**

Because standard views store only the T-SQL query instead of the result set, a standard view might affect database performance in cases in which the T-SQL query in the view involves complex processing of large numbers of rows, multiple levels of joining and data aggregation, repeating patterns of queries, or repeated joins of the same tables on the same keys. In such cases, creating an indexed view might provide better performance than using standard views.

You convert a standard view into an indexed view when you define a unique clustered index (and optional nonclustered indexes) on the view to improve lookup performance. After you create the unique clustered index, the query’s result set is materialized and stored in physical storage, so there is no overhead associated with executing this costly operation at run time. The query optimizer treats an indexed view referenced in the FROM clause as a standard view. However, if designed correctly, the indexed view will be the least expensive path, so SQL Server will use its index to execute the query.

**IMPORTANT Clustered index on a view vs. a table**

There is a big difference between a clustered index on a view and a regular clustered index on a table. The clustered index on a view indexes a result set that might include pre-computed values, aggregates, and data coming from multiple tables.

As you learned in Chapter 4, “Designing a Database for Performance,” adding indexes increases the overhead on the database because the indexes require ongoing maintenance. Therefore, give careful consideration to finding the right balance of indexes and maintenance overhead.

**BEST PRACTICES Use indexed views for infrequently changing data**

Indexed views are recommended for querying infrequently changing data mostly used for read-only purposes. If you need to execute sporadic updates on the data, consider the possibility of dropping any indexed views before the update and re-creating the indexed views after the update to improve update performance.

When designing indexed views, you need to take into account the same factors as when designing table indexes. For example, the indexed view might not provide any significant performance gains if its size is similar to the size of the original table. In addition, when choosing the clustered index key, choose a key compact enough so that its size will not affect performance when creating multiple nonclustered indexes on the view or when doing key comparisons to find a row.

An important benefit of using indexed views in SQL Server 2005 is that the query optimizer might choose to reuse the index on any other query being executed even if the indexed view is not specified in the FROM clause. Database designers need to consider creating indexed views that can satisfy multiple queries, operations, or both.

The query optimizer might consider using the index of an indexed view only when certain conditions are met—for example, when several session options are set to ON (ANSI\_NULLS, ANSI\_WARNINGS, and others), when there is a match between the view index columns and elements in the query, and when the index is the least expensive execution path.

**IMPORTANT Using the NOEXPAND table hint to force index usage**

In SQL Server 2005 Enterprise edition, indexed views are chosen automatically, even when not specified on the FROM clause. However, if you are using a different edition of SQL Server 2005, you need to specify the NOEXPAND table hint to force the use of an index. You might also choose to specify the EXPAND VIEWS hint so that the query optimizer will not use any view indexes.

The following code example uses an indexed view to materialize several aggregated sales data. Notice the different options that need to be set:

SET NUMERIC\_ROUNDABORT OFF;

SET ANSI\_PADDING, ANSI\_WARNINGS, CONCAT\_NULL\_YIELDS\_NULL, ARITHABORT,

QUOTED\_IDENTIFIER, ANSI\_NULLS ON;

GO

CREATE VIEW [Sales].[vTotalSalesForAllRegions]

WITH SCHEMABINDING

AS

SELECT

TerritoryID,

SUM(SubTotal) AS SubTotal,

SUM(TotalDue) AS TotalDue, COUNT\_BIG(\*) AS CountBig

FROM Sales.SalesOrderHeader GROUP BY TerritoryID GO

CREATE UNIQUE CLUSTERED INDEX IDX\_V1

ON [Sales].[vTotalSalesForAllRegions] (TerritoryID);

GO

There are other requirements that need to be fulfilled when designing indexed views. To learn about them, see the “Creating Indexed Views” topic in SQL Server 2005 Books Online.

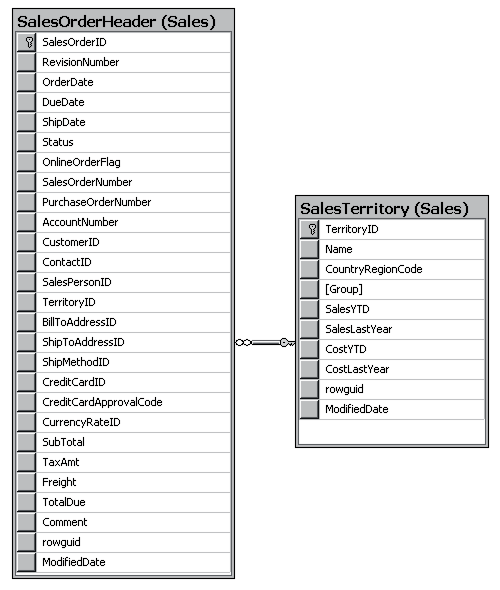
**Designing Partitioned Views**

A partitioned view consists of a T-SQL query that consolidates the data coming from multiple tables, called *member tables*. Each of these member tables has been designed to partition the data horizontally by storing certain ranges of data based on a *partitioning column*. A partitioned view provides a unified view of all the data stored in any number of member tables and hides the complexity of querying (local or remote) database servers to provide a single consolidated view of the data.

The partitioning column holds the values that specify where to look for a range of values. These ranges are enforced through CHECK constraints in the partitioning column. This column can be of any data type (usually numeric types and date types), but it must be part of the table primary key, it cannot accept nulls, and it cannot be an IDENTITY column.

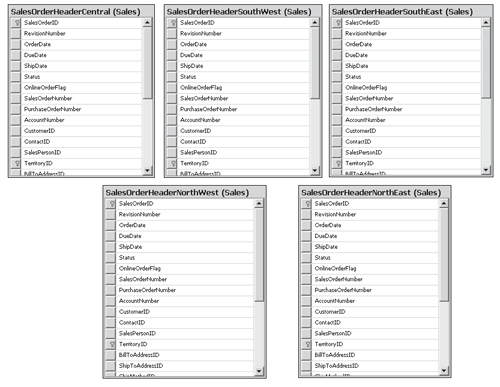
The main difference—and benefit—of using a partitioned view is the declaration of CHECK constraints on each partitioning column to filter the values that can be inserted in each member table. (These values cannot overlap between tables.) When the view is queried, the SQL Server 2005 query optimizer first validates the CHECK constraints on each member table so that only the necessary tables are queried according to the lookup ranges required in the query, resulting in a performance improvement. If no CHECK constraints are defined, this would be the same as having a standard view.

The sample *AdventureWorks* database was designed to run on a single server, so all of the sales information is consolidated in one table called SalesOrderHeader, which has a foreign key constraint to the SalesTerritory table, as Figure 6-1 shows.



**Figure 6-1** The SalesOrderHeader table and SalesTerritory table in *AdventureWorks*

Suppose that instead of consolidating the data in a centralized way like this, you want each of the territories to hold its own sales data. So, you create a SalesOrderHeader table for each territory, as Figure 6-2 shows. (This example creates a SalesOrderHeader table for the Central, Northwest, Northeast, Southwest, and Southeast territories.)



**Figure 6-2** SalesOrderHeader tables for each territory

Each table has been designed as shown in the following code sample:

CREATE TABLE [Sales].[SalesOrderHeaderCentral](

[SalesOrderID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,

[RevisionNumber] [tinyint] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_RevisionNumber] DEFAULT ((0)),

[OrderDate] [datetime] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_OrderDate] DEFAULT (getdate()),

[DueDate] [datetime] NOT NULL,

[ShipDate] [datetime] NULL,

[Status] [tinyint] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_Status] DEFAULT ((1)),

[OnlineOrderFlag] [dbo].[Flag] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_OnlineOrderFlag] DEFAULT ((1)),

[SalesOrderNumber] AS

(isnull(N'SO'+CONVERT([nvarchar](23),[SalesOrderID],0),N'\*\*\* ERROR \*\*\*')),

[PurchaseOrderNumber] [dbo].[OrderNumber] NULL,

[AccountNumber] [dbo].[AccountNumber] NULL,

[CustomerID] [int] NOT NULL,

[ContactID] [int] NOT NULL,

[SalesPersonID] [int] NULL,

<msl\_b>[TerritoryID] [int] NOT NULL CHECK (TerritoryID BETWEEN 1 AND 10), </msl\_b>

[BillToAddressID] [int] NOT NULL,

[ShipToAddressID] [int] NOT NULL,

[ShipMethodID] [int] NOT NULL,

[CreditCardID] [int] NULL,

[CreditCardApprovalCode] [varchar](15) NULL,

[CurrencyRateID] [int] NULL,

[SubTotal] [money] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_SubTotal] DEFAULT ((0.00)),

[TaxAmt] [money] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_TaxAmt] DEFAULT ((0.00)),

[Freight] [money] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_Freight] DEFAULT ((0.00)),

[TotalDue] AS (isnull(([SubTotal]+[TaxAmt])+[Freight],(0))),

[Comment] [nvarchar](128) NULL,

[rowguid] [uniqueidentifier] ROWGUIDCOL NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_rowguid] DEFAULT (newid()),

[ModifiedDate] [datetime] NOT NULL

CONSTRAINT [DF\_SalesOrderHeaderCentral\_ModifiedDate] DEFAULT (getdate()),

CONSTRAINT [PK\_SalesOrderHeaderCentral\_SalesOrderID] PRIMARY KEY CLUSTERED

(

[SalesOrderID] ASC, [TerritoryID]

)

WITH (

PAD\_INDEX = OFF,

STATISTICS\_NORECOMPUTE = OFF,

IGNORE\_DUP\_KEY = OFF,

ALLOW\_ROW\_LOCKS = ON,

ALLOW\_PAGE\_LOCKS = ON

) ON [PRIMARY]

) ON [PRIMARY]

Notice that the TerritoryID column has been declared with a CHECK constraint that filters the values that this table can hold; this table will hold values only for territories with IDs between 1 and 10.

Then, you can create a partitioned view to consolidate the data coming from all the sales territories, as in the following code example:

CREATE VIEW [Sales].[vTotalSalesOrderHeaders]

AS

SELECT \* FROM [Sales].[SalesOrderHeaderCentral]

UNION ALL

SELECT \* FROM [Sales].[SalesOrderHeaderSouthWest]

UNION ALL

SELECT \* FROM [Sales].[SalesOrderHeaderSouthEast]

UNION ALL

SELECT \* FROM [Sales].[SalesOrderHeaderNorthWest]

UNION ALL

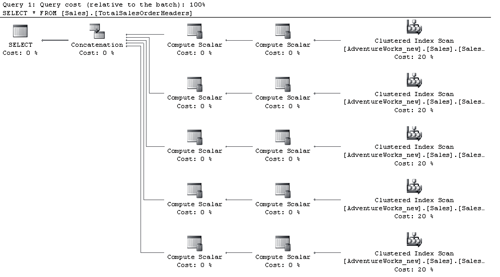
SELECT \* FROM [Sales].[SalesOrderHeaderNorthEast]

In case you want to look at all the sales order headers coming from all territories, you can query the partitioned view with a query such as this:

SELECT \*

FROM [Sales].[vTotalSalesOrderHeaders]

When the view is executed without a WHERE clause, the query optimizer queries each of the member tables, as Figure 6-3 shows.



**Figure 6-3** Estimated query plan for the execution of a partitioned view without a filter

However, if a WHERE clause is declared, the query optimizer validates the CHECK constraints in the partitioning column, so only the necessary tables are queried. For example, to look for the sales order header coming from the territory with ID 15, you would execute a query such as this:

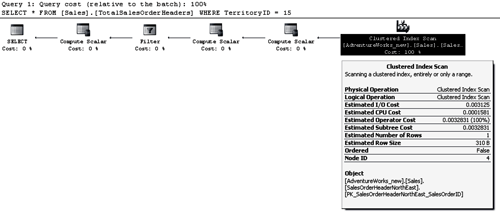
SELECT \*

FROM [Sales].[vTotalSalesOrderHeaders]

WHERE TerritoryID = 15

The query optimizer knows that Territory ID 15 is located in the range of values coming from the SalesOrderHeaderNortheast table.

Look at the difference in the estimated query plan for executing the partitioned view with a filter, which Figure 6-4 shows.



**Figure 6-4** Estimated query plan for the execution of a partitioned view with a filter

When the member tables contained inside the partitioned view are all available inside the same database server, the partitioned view is called a *local partitioned view*. If at least one of the member tables is located on a remote server, it is called a *distributed partitioned view*.

**Federated Database Servers with Partitioned Views**

You can use partitioned views to enhance the performance of a system by horizontally partitioning and spreading the load through a set of remote servers that cooperate to share the processing load. This is a technique called *federated database servers*, which you use to *scale out* a set of servers to support the processing of large systems and Web sites.

**IMPORTANT Scaling out vs. scaling up**

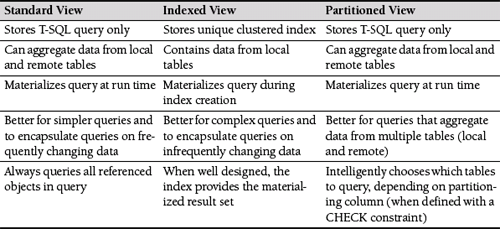
Scaling out is the technique of increasing the processing power of a system by adding one or more additional computers, or nodes, instead of strengthening the hardware of a single computer (to *scale up*).

In a set of federated database servers, no matter which server is queried, they all return the same data. This result is achieved by creating partitioned views in each of the servers that consolidate the local data plus the remote data from the other servers in the federation. Each federated server is known to one another through a linked server declaration.

**IMPORTANT Partitioned views vs. partitioned tables**

As with any other view, partitioned views encapsulate only the T-SQL query used to consolidate the data. The partitioned view does not impose (or care about) any physical distribution of the data. This is the main difference between partitioned views and partitioned tables.

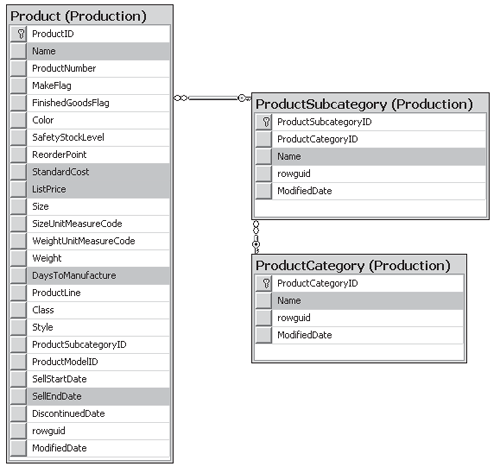
Going back to the previous example that extended the sample *AdventureWorks* database, you distributed the sales information in different member tables (one for each territory) to achieve location independence so that each territory can manage its own SalesOrderHeader table in its own database server. At the same time, however, you provided a consolidated view by declaring the other regions as linked servers and providing a partitioned view that unions all the data, thus creating a federation of database servers. Table 6-1 summarizes some key differences between the three types of views.



**Table 6-1** Types of Views in SQL Server 2005

**Practice: Designing Views**

In this practice, you must apply the concepts from Lesson 1, “Designing Views,” to design a standard view and an indexed view. All the practices in this chapter refer to the Production set of tables from the *AdventureWorks* database, which the entity-relationship diagram (ERD) in Figure 6-5 shows.



**Figure 6-5** Entity-relationship diagram of the Production set of tables from the *AdventureWorks* database

The grayed rows in Figure 6-5 are the ones that will be needed in the following exercises.

**IMPORTANT Practices build upon each other**

The practices in this chapter build upon each other. You should not delete your work after you finish this practice.

**Exercise 1: Create a Standard View**

In this exercise, as the database designer for *AdventureWorks*, you must design a view that returns products that yield the most revenue in terms of the difference between the price and the cost to produce. The production department is looking for products sold at a higher price but produced at a lower cost. Make sure that the view also returns the Category and Subcategory names for the product. Design your own view before reading the suggested answer.

**Suggested Answer**

The following view satisfies the requirements:

USE [AdventureWorks]

GO

CREATE VIEW Get\_Products\_Estimated\_Revenue

AS

SELECT

Production.Product.ProductID,

Production.ProductCategory.Name AS ProductCategory,

Production.ProductSubcategory.Name AS ProductSubCategory,

Production.Product.Name,

Production.Product.ListPrice - Production.Product.StandardCost AS Revenue

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

WHERE (Production.Product.SellEndDate IS NULL)

**Exercise 2: Create an Indexed View**

In this exercise, you decide to convert the preceding standard view into an indexed view to improve performance when reading the ProductID, the product revenue, and the days required to manufacture the product. You execute the following T-SQL declaration to generate the index on the view:

CREATE UNIQUE CLUSTERED INDEX IDX\_V1

ON dbo.Get\_Products\_Estimated\_Revenue (ProductID, Revenue, DaysToManufacture);

What modifications are required in the view’s T-SQL code declaration?

**Suggested Answer**

You need to make three key changes: mark the view declaration with the WITH SCHEMA-BINDING option, include an extra column in the SELECT list returning the *COUNT\_BIG(\*)* function, and include the DaysToManufacture column in the result set.

**Quick Check**

1. What type of view would you create if you needed to pre-aggregate data coming from multiple remote database servers?
2. What is the storage cost of defining a standard view?
3. What is the storage cost of defining an indexed view?

**Quick Check Answers**

1. Either a standard view or a partitioned view would work for pre-aggregating data from multiple remote database servers. Indexed views cannot be created when data is coming from remote tables.
2. There is no storage cost with standard views. Standard views store only the T-SQL query. This T-SQL query is executed every time the view is called, so the most up-to-date data is retrieved.
3. Indexed views materialize the results of the query by creating an index structure. As covered in Chapter 4, the size of the index depends on the type of index and the columns chosen to be part of the index.