

# Views in SQL Server 2000

By: Kristofer Gafvert

## Copyright Information

Copyright © 2003 Kristofer Gafvert ([kgafvert@ilopia.com](mailto:kgafvert@ilopia.com)). No part of this publication may be transmitted, reproduced, or republished in any way, without written permissions by the author. The only website that is allowed to publish this document is ilopia.com, and its sub domains. If this document was downloaded from another website, please contact the author by using the email address above.

If any of these rules are broken, legal actions will be taken for plagiarism. Plagiarism is against the law!

## Table of contents

Copyright Information .....	2
Table of contents .....	3
Introduction .....	4
Advantages of using views .....	5
Create a view .....	6
Create a simple view .....	6
With Encryption .....	8
With Schemabinding .....	9
With View_Metadata .....	11
With Check Option .....	11
Using views .....	12
Retrieve data .....	13
Update data .....	13
Drop views .....	13
Alter views .....	14
Partitioned views .....	14
Indexed views .....	17

## Introduction

A view is quite easy to define, but a lot more difficult to create, use and manage. It's not anymore than a named SELECT statement, or a virtual table. You can select data, alter data, remove data and all other things you can do with a table (with some limitations). But there is a difference between a view and a table. The data accessible through a view is not stored in the database as its own object. It's stored in the underlying tables that make up the view. The only thing that is stored in the database is the SELECT statement that forms the virtual table (with one exception, indexed views, which will not be covered in this article).

ProductID	ProductName	SupplierID	CategoryID
1	Chai	1	1
2	Chang	1	1
3	Aniseed Syrup	1	2
4	Chef Anton's Cajun	2	2
5	Chef Anton's Gumb	2	2
6	Grandma's Boysen	2	2
7	Uncle Bob's Organ	3	1
8	Northwoods Cranb	3	2
9	Mishi Kobe Niku	4	6
10	Ikura	4	8
11	Queso Cabrales	5	4
12	Queso Manchego I	5	4
13	Konbu	6	8
14	Tofu	6	7
15	Genen Shouyu	6	2
16	Pavlova	7	3
17	Alice Mutton	7	6

OrderID	ProductID	UnitPrice	Quantity	Discount
10248	11	14	12	0
10248	42	9.8	10	0
10248	72	34.8	5	0
10249	14	18.6	9	0
10249	51	42.4	40	0
10250	41	7.7	10	0
10250	51	42.4	35	0.15
10250	65	16.8	15	0.15
10251	22	16.8	6	0.05
10251	57	15.6	15	0.05
10251	65	16.8	20	0
10252	20	64.8	40	0.05
10252	33	2	25	0.05
10252	60	27.2	40	0
10253	31	10	20	0
10253	39	14.4	42	0
10253	49	16	40	0
10254	24	3.6	15	0.15
10254	55	19.2	21	0.15

ProductName	Total
Alice Mutton	32695.799819946
Aniseed Syrup	30.74
Boston Cloud Meat	17910.6298923492
Camembert Ferrot	46825.4801330566
Carnarvon Tigers	29171.875
Chai	12788.1000595093
Chang	16355.9600448600
Chèvreuse verte	12294.5399494171
Chef Anton's Cajun Seasonin	8567.89999389648
Chef Anton's Gumbo Mix	5347.20000457764
Chocolate	1368.71252441406
Côte de Blaye	141396.735229492
Escargots de Bourgogne	5881.67504882013
Filo Mix	3232.94999504089
Flotemysost	19551.0250015259
Geitost	1648.125
Genen Shouyu	1784.82499694824

The above picture is an example of a view. It's based on the Products table and Order Details table of the Northwind Sample Database. What we do is join the two tables together, to produce a virtual table, or a view as we will call it from now on. We get the ProductName from the Products table, and we sum UnitPrice, Quantity and Discount

from the Order Details table, to find out how much money every product has given the company. Now, say that we called the view vwProductSales, we could find out the best selling product with this query:

```
SELECT TOP 1 *  
FROM vwProductSales  
ORDER BY Total DESC
```

*Output:*

```
ProductName Total  
Côte de Blaye 141396.73522949219
```

And remember, nothing is actually stored in vwProductSales, but the statement that joins these two tables together.

I haven't told you yet how to create this view, be patient, but as you can see, views are quite handy.

## Advantages of using views

As you could see in the previous section, a view is very useful. Here are some other scenarios when a view can be very useful.

- **Restrict data access and/or simplify data access**  
A view can be used to limit the user to only use a few columns in a table. For example if we do not want a user to be able to access all columns because of security. But it could also be because not all columns are interesting for the user. It is also possible to limit access to rows, using a WHERE clause. If we use USER\_ID(), we can even find out who is using the view, and return only data that is relevant for this user. Furthermore, a view can join several tables, in several databases on several servers, but all the user use is the view's name. Simple, but powerful!
- **Simplify data manipulation**  
We can also use a view to manipulate data. Often with a relational database design, this means that many tables must be joined together. Using a view can simplify this, and the users do not need to know all tables involved.
- **Import and export data**  
A view can also be used to export and/or import data to/from another application. Both the bcp utility and BULK INSERT works with a view.
- **Merge data**  
A so called Partition View can be used to merge data from multiple tables in multiple databases, so it appears as one table only, but it is in fact several tables. This can be accomplished by using the UNION operator. For example if we had customers in Europe and United States, we could have one server for Europe, and

one for United States, with tables that are identical, and then merge all data in a partitioned view. More on this later.

## Create a view

Now when we know what a view is, and when it's a good idea to use one, let's create one. And if you thought it was difficult, you will get surprised. To create a view, you actually don't have to know more than how to select data from a table.

First the syntax:

```
CREATE VIEW [< owner >.] view_name [ ( column [ , ...n ] ) ]
[ WITH < view_attribute > [ , ...n ] ]
AS
select_statement
[ WITH CHECK OPTION ]

< view_attribute > ::=
    { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }
```

## Create a simple view

Before we create our first view, we should know some rules about the select statement. It cannot:

- Include ORDER BY clause, unless there is also a TOP clause (remember that a view is nothing else but a virtual table, why would we want to order it?)
- Include the INTO keyword.
- Include COMPUTE or COMPUTE BY clauses.
- Reference a temporary table or a table variable.

Also, not everyone can create a view. You must be a member of the fixed database roles **db\_owner** or **db\_ddladmin**. Or, a member of the **sysadmin** server role, or **db\_owner** database role must have given you permissions to create a view by running this:

```
GRANT CREATE VIEW TO [SqlServer01\Kristofer]
```

The above statement will grant Kristofer on the server SqlServer01 permissions to create views.

I thought it would be a good idea to create the vwProductSales I used earlier as an example, and that is also the one you can see in the picture in the Introduction section. So, start Query Analyzer, and logon as a user with permissions to create a view. Then run this:

```
USE Northwind
GO
CREATE VIEW vwProductSales
```

```
AS
SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
GO
```

*Output:*

The command(s) completed successfully.

As you can see, it's not difficult at all! It's basically a SELECT statement.

Now, say that we by some reason want this ordered by the Total column. Again, think about a view as a virtual table, so there is actually no reason to do it (do you usually sort your data in your tables when you insert data? Me neither!), but just for fun:

```
USE Northwind
GO
CREATE VIEW vwProductSalesSorted
AS
SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
ORDER BY Total DESC
GO
```

*Output:*

Server: Msg 1033, Level 15, State 1, Procedure vwProductSalesSorted, Line 7  
The ORDER BY clause is invalid in views, inline functions, derived tables, and subqueries, unless TOP is also specified.

As you can see by the output, we were not allowed to do this. On the other hand, if we were only interested in the best selling products, we could write this:

```
USE Northwind
GO
CREATE VIEW vwProductSalesTop10
AS
SELECT TOP 10 Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS
Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
ORDER BY Total DESC
GO
```

*Output:*

The command(s) completed successfully.

As you can see, this worked. We did not violate the rules of a view, and this was also one of the advantages of using a view, to select only the data we were interested in.

I've told you several times that a view is a virtual table, and the only thing that is stored in the database is the select statement. It is in fact stored in the **Syscomments** system table. And to prove this, run this statement:

```
USE Northwind
```

```
GO
SELECT [text]
FROM syscomments
WHERE id = OBJECT_ID('vwProductSales')
```

**Output:**

```
CREATE VIEW vwProductSales
AS
    SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS
Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
```

(1 row(s) affected)

**Note:**

If you did not get the same result, it might be because Query Analyzer is configured to return results to grids. You can change it to return result to text from **Tools->Options**, and then click on the **Results** tab.

As you can see, there it is, our view! But of course, Microsoft do not force us to write all this to get information about a view (or a rule, a default, an unencrypted stored procedure, user-defined function, trigger). The system stored procedure `sp_helptext` does this for us.

```
EXEC sp_helptext vwProductSales
```

The same goes for `sp_depends`, which gives us information about what other objects our view depends on.

```
EXEC sp_depends vwProductSales
```

**Output:**

Name	type	updated	selected	column
dbo.Order Details	user table	no	no	UnitPrice
dbo.Order Details	user table	no	no	Quantity
dbo.Order Details	user table	no	no	Discount
dbo.Products	user table	no	no	ProductID
dbo.Products	user table	no	no	ProductName
dbo.Order Details	user table	no	no	ProductID

## With Encryption

As we saw, it is very simple to see the actually code for the view. Usually, this is fine, but say that you develop an e-commerce system, probably using a database, with some views. You ship it to your customer, but you do not want them to be able to see the code for the view. Then you can use the `ENCRYPTION` attribute. In fact, all objects containing code can be encrypted. So, let's create an encrypted version of our view above.

```
USE Northwind
GO
CREATE VIEW vwProductSalesEncrypted
```

```
WITH ENCRYPTION
AS
    SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
    FROM [Order Details] AS OD
    INNER JOIN Products ON OD.ProductID = Products.ProductID
    GROUP BY Products.ProductName
GO
```

*Output:*

The command(s) completed successfully.

If we now try to query the text of this view, we will not get back the code.

```
EXEC sp_helptext vwProductSalesEncrypted
```

*Output:*

The object comments have been encrypted.

Now you think that this is secure, and your customer will never be able to find out the code for our encrypted view. You couldn't be more wrong. But it is beyond the scope of this article to show you how to decrypt the view; I just wanted to tell you that it is possible.

## With Schemabinding

Wondered what would happen if we altered one of the underlying tables so a column that was referenced in the view no more existed? If we do not specify SCHEMABINDING, we could delete a column, and our view would fail when we later used it.

```
USE Northwind
GO
SELECT * INTO Employees2 FROM Employees
GO
CREATE VIEW vwEmployeeAge ([Name], Age)
AS
    SELECT TitleOfCourtesy + ' ' + FirstName + ' ' + ' ' + LastName,
           DATEDIFF(yy, BirthDate, GETDATE())
    FROM Employees2
GO
ALTER TABLE Employees2 DROP COLUMN FirstName
SELECT * FROM vwEmployeeAge
```

*Output:*

Server: Msg 207, Level 16, State 3, Procedure vwEmployeeAge, Line 3  
Invalid column name 'FirstName'.

Server: Msg 4413, Level 16, State 1, Line 1

Could not use view or function 'vwEmployeeAge' because of binding errors.

I think I have to explain some of the code first (note that we did not specify SCHEMABINDING in this example). On line three, we create a new table called Employees2, and inserts all data into it (so we do not have to modify the original table in the Northwind database). Then we create the view with the name vwEmployeeAge. We also specify the name of the columns, which we have never done before. This is valid, and in fact, we must do it here, since otherwise the columns would have no name. We

could however use AS to specify the name in the SELECT query, but to be honest, I like this other way better.

Anyway, after we have created the view, we drop the column FirstName from Employees2, and then we try to use the view (I know, I haven't showed you how to use a view yet, be patient, it will be in the next section). As you can see, it doesn't work very well to use this view.

So, to protect ourselves from this in the future, we can use SCHEMABINDING. First of all, drop the table Employees2, and re-create it.

```
USE Northwind
GO
DROP TABLE Employees2
GO
SELECT * INTO Employees2 FROM Employees
GO
```

Then create the view with SCHEMABINDING.

```
USE Northwind
GO
CREATE VIEW vwEmployeeAgeSchemabinding ([Name], Age)
WITH SCHEMABINDING
AS
    SELECT TitleOfCourtesy + ' ' + FirstName + ' ' + ' ' + LastName,
           DATEDIFF(yy, BirthDate, GETDATE())
    FROM Employees2
GO
```

*Output:*

Server: Msg 4512, Level 16, State 3, Procedure vwEmployeeAgeSchemabinding, Line 4  
Cannot schema bind view 'vwEmployeeAgeSchemabinding' because name 'Employees2' is  
invalid for schema binding. Names must be in two-part format and an object cannot reference  
itself.

Ooops, didn't work did it? When using SCHEMABINDING, the select statement must include the two-part names (owner.object) of tables, views or user-defined functions referenced. So, let's try this instead:

```
USE Northwind
GO
CREATE VIEW vwEmployeeAgeSchemabinding ([Name], Age)
WITH SCHEMABINDING
AS
    SELECT TitleOfCourtesy + ' ' + FirstName + ' ' + ' ' + LastName,
           DATEDIFF(yy, BirthDate, GETDATE())
    FROM dbo.Employees2
GO
```

There we go, much better. Now try to drop the column FirstName again.

```
ALTER TABLE Employees2 DROP COLUMN FirstName
```

*Output:*

Server: Msg 5074, Level 16, State 3, Line 1  
The object 'vwEmployeeAgeSchemabinding' is dependent on column 'FirstName'.

Server: Msg 4922, Level 16, State 1, Line 1

ALTER TABLE DROP COLUMN FirstName failed because one or more objects access this column.

As you can see, we cannot drop the column now.

## With View\_Metadata

If WITH VIEW\_METADATA is specified, SQL Server will return the metadata information about the view, instead of the underlying tables to the DBLIB, ODBC and OLE DB APIs, when metadata is being requested. Metadata is information about the view's properties, for example column names or type. If WITH VIEW\_METADATA is used, the client application is able to create an updateable client side cursor, based on the view.

## With Check Option

This is probably the most useful of all options. It guarantees that data modification through the view complies with the WHERE condition. This means that if you insert or update data, it is not possible that it "disappears" when we query the view afterwards. I think this is explained best with an example.

```
USE Northwind
GO
CREATE VIEW vwCustomersInSweden
AS
    SELECT *
    FROM Customers
    WHERE Country = 'Sweden'
GO
SELECT * FROM vwCustomersInSweden
```

*Output:*

<b>CustomerID</b>	<b>CompanyName</b>	<b>ContactName</b>	<b>...</b>
BERGS	Berglunds snabbköp	Christina Berglund	...
FOLKO	Folk och fä HB	Maria Larsson	...

Now, let's insert data using this view, and then try to find what we just inserted.

```
INSERT INTO vwCustomersInSweden
VALUES
(
    'SHOPS',
    'Shop Shop Shop',
    'John Doe',
    'Owner',
    '134 Polk St. Suite 5',
    'San Francisco',
    'CA',
    '94117',
    'USA',
    '(415) 555-1234',
    NULL
)
```

```
)  
SELECT * FROM vwCustomersInSweden
```

*Output:*

<b>CustomerID</b>	<b>CompanyName</b>	<b>ContactName</b>	<b>...</b>
BERGS	Berglunds snabbköp	Christina Berglund	...
FOLKO	Folk och få HB	Maria Larsson	...

Yes, it's true, it's not there. When we created this view, we probably not only wanted to limit the user to only see the Swedish customers. We probably also wanted to deny the user to insert data for other countries but Sweden. To accomplish this, we have to use **WITH CHECK OPTION**.

```
USE Northwind  
GO  
CREATE VIEW vwCustomersInSwedenCheck  
AS  
    SELECT *  
    FROM Customers  
    WHERE Country = 'Sweden'  
WITH CHECK OPTION  
GO
```

If we try to insert the same data again (we first delete it), we will get an error.

```
DELETE FROM Customers WHERE CustomerID = 'SHOPS'  
INSERT INTO vwCustomersInSwedenCheck  
VALUES  
(  
    'SHOPS',  
    'Shop Shop Shop',  
    'John Doe',  
    'Owner',  
    '134 Polk St. Suite 5',  
    'San Francisco',  
    'CA',  
    '94117',  
    'USA',  
    '(415) 555-1234',  
    NULL  
)
```

*Output:*

Server: Msg 550, Level 16, State 1, Line 1  
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.  
The statement has been terminated.

## Using views

You can use a view to retrieve data, or update data. A view can be used whenever a table could be used, for retrieving data. For updating data, this may not always be true, and it depends on the view.

## Retrieve data

We have already seen several examples of retrieving data through a view. I will now compare two queries, one where we use a view, and one without.

```
SELECT TOP 1 *
FROM vwProductSales
ORDER BY Total DESC
```

*Output:*

<b>ProductName</b>	<b>Total</b>
Côte de Blaye	141396.73522949219

```
SELECT
    TOP 1 Products.ProductName,
    SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
ORDER BY Total DESC
```

*Output:*

<b>ProductName</b>	<b>Total</b>
Côte de Blaye	141396.73522949219

Now, which one of these two queries do you find most simple to use? The first one? Yeah, though so!

## Update data

There are some restrictions for updating data through a view.

- A view cannot modify more than one table. So if a view is based on two or more tables, and you try to run a DELETE statement, it will fail. If you run an UPDATE or INSERT statement, all columns referenced in the statement must belong to the same table.
- It's not possible to update, insert or delete data in a view with a DISTINCT clause.
- You cannot update, insert or delete data in a view that is using GROUP BY.
- It's not possible to update, insert or delete data in a view that contains calculated columns.

Also be aware of columns that cannot contain NULL. If it has no default value, and is not referenced in the view, you will not be able to update the column.

## Drop views

Deleting a view is not complicated. It's as easy as:

```
DROP VIEW vwProductSalesTop10
```

## Alter views

Alter views is as simple as deleting them. We can try to remove the encryption of the view vwProductSalesEncrypted

```
ALTER VIEW vwProductSalesEncrypted
AS
    SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
    FROM [Order Details] AS OD
    INNER JOIN Products ON OD.ProductID = Products.ProductID
    GROUP BY Products.ProductName
GO
```

*Output:*

The command(s) completed successfully.

## Partitioned views

A partitioned view uses the UNION ALL operator to merge all member tables (which must be structured in the same way). The tables can be stored in the same SQL Server, or in multiple SQL Servers. It must however be clear what data belongs to each partition, by using CHECK constraints and data cannot overlap (so you cannot have one table with customers ID from 1 to 50, and another table with customer ID from 25 to 75). For example, say that we have three different kinds of customers, those from Sweden, those from UK, and those from USA. The number of rows in the customers table is increasing very fast, so we would like to split the existing table into three other tables, on three different servers (in my example, we will do this on one server only, so as many of you as possible can try it). We then create a view on each server, to access information about all customers. But, this is very smart, if we later use the view to access data from two servers (because the WHERE clause limits it), it will not transfer data from the third server.

Before creating the tables, it is a good idea to mention the rules for a so called partitioning column (the column(s) that makes it impossible to overlap data).

- It is not a calculated, identity, default or timestamp column
- It cannot have the value NULL
- It is part of the primary key
- It should be validated by a CHECK constraint, but cannot be validated using these operators <> and !
- Only one CHECK constraint exists on the partitioning column.

There are also some rules the tables need to follow:

- The primary key should be defined on the same columns
- The table cannot have indexes created on computed columns
- All tables should have the same ANSI padding setting.

Now, finally, let's create the three tables (which we will in this example place in the same database (Northwind) on the same server).

```
SELECT * INTO CustomersSweden
FROM Customers
WHERE Country = 'Sweden'
GO
ALTER TABLE CustomersSweden ALTER COLUMN
Country NVARCHAR(15) NOT NULL
GO
ALTER TABLE CustomersSweden ADD
CONSTRAINT PK_CustomersSweden PRIMARY KEY (CustomerID, Country),
CONSTRAINT CK_CustSweden_Country CHECK (Country IN ('Sweden'))
GO

SELECT * INTO CustomersUK
FROM Customers
WHERE Country = 'UK'
GO
ALTER TABLE CustomersUK ALTER COLUMN
Country NVARCHAR(15) NOT NULL
GO
ALTER TABLE CustomersUK ADD
CONSTRAINT PK_CustomersUK PRIMARY KEY (CustomerID, Country),
CONSTRAINT CK_CustUK_Country CHECK (Country IN ('UK'))
GO

SELECT * INTO CustomersUSA
FROM Customers
WHERE Country = 'USA'
GO
ALTER TABLE CustomersUSA ALTER COLUMN
Country NVARCHAR(15) NOT NULL
GO
ALTER TABLE CustomersUSA ADD
CONSTRAINT PK_CustomersUSA PRIMARY KEY (CustomerID, Country),
CONSTRAINT CK_CustUSA_Country CHECK (Country IN ('USA'))
GO
```

This script creates three different tables (CustomersSweden, CustomersUK and CustomersUSA), all based on the Customers table in the Northwind database. But as you remember, a partitioning column cannot be NULL, so we have to alter the column Country so it does not accept NULL values. We also have to add a PRIMARY KEY constraint, and a check constraint.

After we have created the tables, we can create the partitioned view:

```
CREATE VIEW vwCustomers
AS
    SELECT * FROM CustomersSweden
    UNION ALL
    SELECT * FROM CustomersUK
    UNION ALL
    SELECT * FROM CustomersUSA
GO
```

If we now select all data from this view, it looks like it's one table, with customers from USA, Sweden and UK.

```
SELECT * FROM vwCustomers
```

*Output:*

<b>CustomerID</b>	<b>CompanyName</b>	<b>...</b>	<b>Country</b>	<b>...</b>
BERGS	Berglunds snabbköp	...	Sweden	...
FOLKO	Folk och få HB	...	Sweden	...
AROUT	Around the Horn	...	UK	...
...	...	...	...	...

But, what happens if we insert data using this view?

```
INSERT INTO vwCustomers
VALUES
(
    'SHOPS',
    'Shop Shop Shop',
    'John Doe',
    'Owner',
    '111 East 12th',
    'New York',
    'NY',
    '94117',
    'USA',
    '(415) 555-1234',
    NULL
)
```

Yep, it's John Doe again! Now try this, and see where you find John Doe.

```
SELECT * FROM CustomersSweden
SELECT * FROM CustomersUK
SELECT * FROM CustomersUSA
```

You also found him in the table CustomersUSA? And we do not even have to know that this table exists! He ends up there because that is the only table he will not violate the CHECK constraint in.

Say that John Doe moves to UK, but everything else is the same! (because we are lazy :-)

```
UPDATE vwCustomers
SET Country = 'UK'
WHERE CustomerID = 'SHOPS'
```

**Note:**

Didn't work? Only the Developer and Enterprise Edition of SQL Server 2000 allow INSERT, UPDATE and DELETE operations on partitioned views.

In which table do you think it's stored this time? Yep, that is correct, in the CustomersUK table.

But, as always, there are some rules about updating data through a partitioned view.

- The INSERT statement must supply values for all columns in the view. It doesn't matter if the underlying tables have a DEFAULT constraint, or allow NULLs.
- It must satisfy at least one of the underlying constraints (we cannot insert a customer from France in the example above).
- The DEFAULT keyword does not work in INSERT and UPDATE statements.
- Columns that are IDENTITY columns in one or more member tables cannot be modified.
- Data cannot be modified if one of the tables contains a timestamp column.
- If there is a self-join with the same view or with any of the member tables in the statement, INSERT, UPDATE and DELETE do not work.

## Indexed views

It is beyond the scope of this article, but I thought it would be good to at least mention it. As you know now, a view is usually no more than a stored query. Using a view is not always fast, and for example when you must do some calculating based on a few millions rows, it can be very slow. An indexed view will however fix this, at least make it a little bit faster, because it will store the result set in the database, which is updated dynamically. The disadvantage of indexed views is that it will slow down a query that updates data in the underlying tables. You can find more information about indexed views in Books Online.