Partitioned views

I must say I am a big fan of views. They give flexibility of changing the logic of the data retrieval without affecting DAL layers in your application. I posted series of articles about [updateable views](http://usepubs.blogspot.com/2010/06/updateable-views-how-to-use-them.html) and [triggers on views](http://usepubs.blogspot.com/2010/06/instead-of-triggers-on-views.html), their [application in SSIS](http://usepubs.blogspot.com/2010/07/instead-of-triggers-using-views-in-ssis.html) for example.

Recently we finished development of system that loads data from several external sources, makes some calculations on the data and sends data over FTP to downstream systems. I designed the whole data interface to use views and stored procedures. This way, even though we used ORM to perform some of the activities, it was not accessing bare tables. This is good, because you can implement simpler and more flexible security model. The other advantage is that you if you use some simple techniques, you can load data to table and then expose the new data through a view without exposing the logic to the upper layers of your application.

It is quite common requirement to be able to maintain multiple versions of data within table. The requirement is also that data import should not affect other processes which read current version of data. With views it is really simple to achieve. There are many ways of doing this, but let’s have a look at partitioned views technique.

Partitioned view is a view that selects data from more than one table. These tables have to follow specific requirements. Amongst the other, there has to be so called partitioning column defined, which has trusted CHECK constraint defined on it. The tables have to be joined by UNION ALL operator. If you want the view to be updateable, the partitioning column has to be part of the primary key of all tables belonging to the view. And you have to select all columns from the table.

OK, but what is the purpose of creating partitioned views? The short answer is performance. Tables tend to grow bigger and bigger. It becomes more and more expensive to run index updates, backups, and queries against them. If you are able to partition your data and you do it in a right way, you can alleviate some of the issues related to big tables. Classic example would be partitioning by date – you can keep month worth of data in each table and span the view over 12 tables to cover the whole year. Nice thing about partitioned views is that you can index underlying tables in a different way, so for example history tables have more indexes allowing for faster data retrieval, while current month table can be optimized for faster inserts and updates.

Let’s have a look at this scenario. I will use as an example table Sales.CurrencyRate from AdventureWorks database. This table contains 13532 rows in my database, but let’s imagine we have 135 million rows.

When I run this query, I get 37 rows spanning years 2001 - 2004:

1: select count(\*) NumberOfRates, Year([CurrencyRateDate]) RateYear, Month([CurrencyRateDate]) RateMonth

2: from Sales.CurrencyRate

3: group by Year([CurrencyRateDate]), Month([CurrencyRateDate])

4: order by RateYear, RateMonth

We will create four tables, each for each year that is represented by CurrencyRateDate column.

A slightly abbreviated DDL for the Sales.CurrencyRate table is as follows:

1: CREATE TABLE [Sales].[CurrencyRate](

2: [CurrencyRateID] [int] IDENTITY(1,1) NOT NULL,

3: [CurrencyRateDate] [datetime] NOT NULL,

4: [FromCurrencyCode] [nchar](3) COLLATE Latin1\_General\_CI\_AS NOT NULL,

5: [ToCurrencyCode] [nchar](3) COLLATE Latin1\_General\_CI\_AS NOT NULL,

6: [AverageRate] [money] NOT NULL,

7: [EndOfDayRate] [money] NOT NULL,

8: [ModifiedDate] [datetime] NOT NULL CONSTRAINT [DF\_CurrencyRate\_ModifiedDate] DEFAULT (getdate()),

9: CONSTRAINT [PK\_CurrencyRate\_CurrencyRateID] PRIMARY KEY CLUSTERED

10: (

11: [CurrencyRateID] ASC

12: )WITH (IGNORE\_DUP\_KEY = OFF) ON [PRIMARY]

13: ) ON [PRIMARY]

If we want to have updateable partitioned view, we need to modify the constraint to include date part, which will allow the execution engine to identify partition that has to be modified during DML operation. The easiest is to add the CurrencyRateDate column to the primary key.

Let’s create four tables then, one for each year of the data. I will not place here whole DDL for all tables, you can find it in the attached file. This is however the first partition table:

1: CREATE TABLE [Sales].[CurrencyRate\_2001](

2: [CurrencyRateID] [int] IDENTITY(1,1) NOT NULL,

3: [CurrencyRateDate] [datetime] NOT NULL,

4: [FromCurrencyCode] [nchar](3) COLLATE Latin1\_General\_CI\_AS NOT NULL,

5: [ToCurrencyCode] [nchar](3) COLLATE Latin1\_General\_CI\_AS NOT NULL,

6: [AverageRate] [money] NOT NULL,

7: [EndOfDayRate] [money] NOT NULL,

8: [ModifiedDate] [datetime] NOT NULL CONSTRAINT [DF\_CurrencyRate\_2001\_ModifiedDate] DEFAULT (getdate()),

9: constraint CHK\_CurrencyRateDate\_2001 CHECK (CurrencyRateDate>= '20010101' and CurrencyRateDate < '20020101'),

10: CONSTRAINT [PK\_CurrencyRate\_2001\_CurrencyRate\_2001ID] PRIMARY KEY CLUSTERED

11: (

12: [CurrencyRateID] ASC,

13: [CurrencyRateDate] ASC

14: )

15: ) ON [PRIMARY]

As you see above, the CurrencyRateDate column has been included into the primary key of the table and I added constraint CHK\_CurrencyRateDate\_2001 that will take care of the range of dates that can exist in the table. You can use inequality constraint if you’d like to store rows for dates from 2001 or older. Tables for 2002, 2003 and 2004 contain constraints disallowing inserting data outside of the ranges, similar to the the constraint shown above.

I removed identity column from partition tables to make the view updateable – this is one of requirements.

The view is defined as follows:

1: create view Sales.vCurrencyRate

2: as

3: select \* from Sales.CurrencyRate\_2001

4: union all

5: select \* from Sales.CurrencyRate\_2002

6: union all

7: select \* from Sales.CurrencyRate\_2003

8: union all

9: select \* from Sales.CurrencyRate\_2004

Let’s insert some data to the view – that is, the underlying partition tables:

1: insert Sales.vCurrencyRate

2: (

3: [CurrencyRateID],

4: [CurrencyRateDate],

5: [FromCurrencyCode],

6: [ToCurrencyCode],

7: [AverageRate],

8: [EndOfDayRate],

9: [ModifiedDate]

10: )

11: select [CurrencyRateID],

12: [CurrencyRateDate],

13: [FromCurrencyCode],

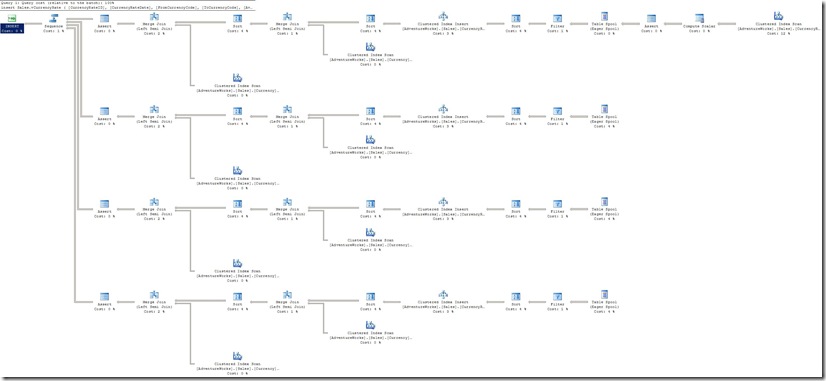
14: [ToCurrencyCode],

15: [AverageRate],

16: [EndOfDayRate],

17: [ModifiedDate] from Sales.CurrencyRate

This is the execution plan created for the insert:



As you see, database engine split data amongst all partition tables according to the partitioning column constraints. This way, you can easily move data from one table to several smaller partitions. Of course, you wouldn’t want to run simple insert statement as above for 135 million rows, but you get the idea.

Partitioned views can be very useful, but apart from the conditions that I listed earlier in this article, there are some catches that might bite you and you have to carefully ponder on your design if you want to make them updateable.

These are some of these gotchas:

Tables referenced in updateable partitioned views cannot have identity column. If they have, you will get following error message:

Msg 4433, Level 16, State 4, Line 3   
Cannot INSERT into partitioned view 'AdventureWorks.Sales.vCurrencyRate' because table '[AdventureWorks].[Sales].[CurrencyRate\_2001]' has an IDENTITY constraint.

Tables cannot have computed columns if you want to have updateable partitioned view. If you try to insert rows into view on tables that have such columns, the error message is following:

Msg 4406, Level 16, State 1, Line 1   
Update or insert of view or function 'Sales.vSalesOrderHeader' failed because it contains a derived or constant field.

The partitioning column has to be defined using equality or inequality operators. For example, you cannot use datepart(year, …) to defined date range for the column, you have to use either BETWEEN.. AND.. or >=.. < operator. Otherwise the error message will be as follows:

Msg 4436, Level 16, State 12, Line 3   
UNION ALL view 'AdventureWorks.Sales.vCurrencyRate' is not updatable because a partitioning column was not found.

The partitioned views do not have to be updateable. You can design tables with computed columns, IDENTITY columns and still use partitioned views to offload certain queries. One of the scenarios is for example daily upload of data. You can create a staging table which will accept the data and after ETL is finished, you alter view definition to include new table and possibly exclude one with older data.

I attach the script for your convenience. You can run it on AdventureWorks database or use it as a template for your code.

If you want to maximize query efficiency, exploit the new optimization capabilities introduced with SQL Server 2000, and allow updates directly to the view, then you need to ensure that the following conditions are met in each of the tables:

* The partitioning column needs to be part of the primary key of the underlying tables, must not allow NULLs, and can't be a computed column.
* The CHECK constraint defined on the partitioning column that defines the partitioning criteria can use only the following operators: BETWEEN, AND, OR, <, <=, >, >=, =.
* The tables can't have identity or timestamp columns, and none of the columns in any of the tables can have a DEFAULT constraint.

As the partitioned tables do, the views need to meet a few conditions to be updateable and to exploit the new optimizing capabilities that distributed partitioned views feature:

* The view can't reference any table or column more than once.
* Each SELECT list must reference all the columns participating in the primary key of the underlying tables.
* The columns in the same ordinal position in the select list in all the select statements must be of exactly the same data type, precision, scale, and collation. And the partitioning column must be in the same ordinal position in all the select statements.
* If a column exists in the base table but doesn't exist in the select list in the view, the column must allow NULLs.

**Collapse image**[SYMPTOMS](javascript:void(0);)

A distributed partitioned view joins horizontally partitioned data from a set of member tables across one or more SQL Servers, which makes the data appear as if it is from one table. If you run any statements that modify the data in the partitioned view after you first disable and then enable the check constraint on any of the involved base tables, you may receive the following error message:

Msg 4436, Level 16, State 12, Line 1  
UNION ALL view 'v1' is not updatable because a partitioning column was not found.

**Collapse image**[CAUSE](javascript:void(0);)

When you disable a check constraint, SQL Server marks the constraint as untrusted, which means that the server does not check for consistency when a user modifies rows that may violate the check constraint.  
  
Later, when this check constraint is enabled, SQL Server maintains its status as untrusted because it cannot validate the data for that column. As a result, the Query Optimizer does not use these constraints to prune out unwanted branches in the partitioned view or treat such columns as range partitioned columns. This behavior is by design.

**Collapse image**[WORKAROUND](javascript:void(0);)

To work around this problem, drop and then re-create the check constraint that was altered. You can use the following query to identify the untrusted check contraints in a database:

SELECT name, status FROM sysobjects WHERE xtype = 'C' AND status &0x800 = 0x800

<http://technet.microsoft.com/en-us/library/aa175850(v=sql.80).aspx>

Msg 4436, Level 16, State 12, Line 71

UNION ALL view 'eCustoms.dbo.vDSX\_SP\_NPL' is not updatable because a partitioning column was not found.

- gia tri contraint trong bang cua cac server phai khac nhau

ex:

eCustoms\_his1.DSX\_SP\_NPL [MA\_HQ]='A01B' OR [MA\_HQ]='P01L'

eCustoms\_his1.DSX\_SP\_NPL [MA\_HQ]='A01B01'

- drop and re-create constraint