What is a database table partitioning?

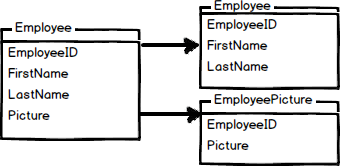
Partitioning is the database process where very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables, queries that access only a fraction of the data can run faster because there is less data to scan.

The main of goal of partitioning is to aid in maintenance of large tables and to reduce the overall response time to read and load data for particular SQL operations.

Vertical Partitioning on SQL Server tables

Vertical table partitioning is mostly used to increase SQL Server performance especially in cases where a query retrieves all columns from a table that contains a number of very wide text or BLOB (varbinary) columns. In this case to reduce access times the BLOB columns can be split to its own table. Another example is to restrict access to sensitive data e.g. passwords, salary information etc. Vertical partitioning splits a table into two or more tables containing different columns:

varbinary(max): hexacode (we can store image , files, folder in this column)



An example of vertical partitioning

An example for vertical partitioning can be a large table with reports for employees containing basic information, such as report name, id, number of report and a large column with report description.

Assuming that ~95% of users are searching on the part of the report name, number, etc. and that only ~5% of requests are opening the reports description field and looking to the description.

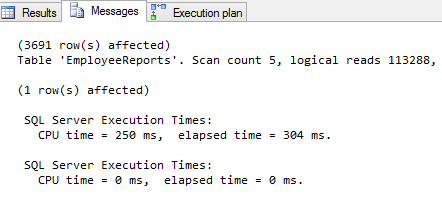
Let’s assume that all those searches will lead to the clustered index scans and since the index scan reads all rows in the table the cost of the query is proportional to the total number of rows in the table and our goal is to minimize the number of IO operations and reduce the cost of the search.

Let’s see the example on the EmployeeReports table:

|  |
| --- |
| CREATE TABLE EmployeeReports  (  ReportID int IDENTITY (1,1) NOT NULL,  ReportName varchar (100),  ReportNumber varchar (20),  ReportDescription varchar (max)  CONSTRAINT EReport\_PK PRIMARY KEY CLUSTERED (ReportID)  )    DECLARE @i int  SET @i = 1    BEGIN TRAN  WHILE @i&lt;100000  BEGIN  INSERT INTO EmployeeReports  (  ReportName,  ReportNumber,  ReportDescription  )  VALUES  (  'ReportName',  CONVERT (varchar (20), @i),  REPLICATE ('Report', 1000)  )  SET @i=@i+1  END  COMMIT TRAN  GO |

If we run a SQL query to pull ReportID, ReportName, ReportNumber data from the EmployeeReports table the result set that a scan count is 5 and represents a number of times that the table was accessed during the query, and that we had 113,288 logical reads that represent the total number of page accesses needed to process the query:

|  |
| --- |
| SET STATISTICS IO ON  SET STATISTICS TIME ON  SELECT er.ReportID, er.ReportName, er.ReportNumber  FROM dbo.EmployeeReports er  WHERE er.ReportNumber LIKE '%33%'  SET STATISTICS IO OFF  SET STATISTICS TIME OFF |



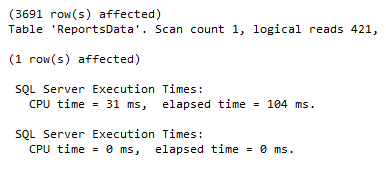
As indicated, every page is read from the data cache, whether or not it was necessary to bring that page from disk into the cache for any given read. To reduce the cost of the query we will change the SQL Server database schema and split the EmployeeReports table vertically.

Next we’ll create the ReportsDesc table and move the large ReportDescription column, and the ReportsData table and move all data from the EmployeeReports table except the ReportDescriptioncolumn:

|  |
| --- |
| CREATE TABLE ReportsDesc  ( ReportID int FOREIGN KEY REFERENCES EmployeeReports (ReportID),    ReportDescription varchar(max)    CONSTRAINT PK\_ReportDesc PRIMARY KEY CLUSTERED (ReportID)  )    CREATE TABLE ReportsData  (  ReportID int NOT NULL,  ReportName varchar (100),  ReportNumber varchar (20),    CONSTRAINT DReport\_PK PRIMARY KEY CLUSTERED (ReportID)  )  INSERT INTO dbo.ReportsData  (      ReportID,      ReportName,      ReportNumber  )  SELECT er.ReportID,  er.ReportName,  er.ReportNumber  FROM dbo.EmployeeReports er |

The same search query will now give different results:

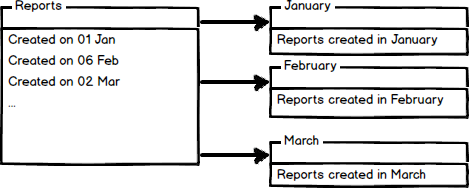
|  |
| --- |
| SET STATISTICS IO ON  SET STATISTICS TIME ON  SELECT er.ReportID, er.ReportName, er.ReportNumber  FROM ReportsData er  WHERE er.ReportNumber LIKE '%33%'  SET STATISTICS IO OFF  SET STATISTICS TIME OFF |



Vertical partitioning on SQL Server tables may not be the right method in every case. However, if you have, for example, a table with a lot of data that is not accessed equally, tables with data you want to restrict access to, or scans that return a lot of data, vertical partitioning can help.

Horizontal Partitioning on SQL Server tables

Horizontal partitioning divides a table into multiple tables that contain the same number of columns, but fewer rows. For example, if a table contains a large number of rows that represent monthly reports it could be partitioned horizontally into tables by years, with each table representing all monthly reports for a specific year. This way queries requiring data for a specific year will only reference the appropriate table. Tables should be partitioned in a way that queries reference as few tables as possible.



Tables are horizontally partitioned based on a column which will be used for partitioning and the ranges associated to each partition.

Partitioning column is usually a datetime column but all data types that are valid for use as index columns can be used as a partitioning column, except a timestamp column.

The ntext, text, image, xml, varchar(max), nvarchar(max), or varbinary(max), Microsoft .NET Framework common language runtime (CLR) user-defined type, and alias data type columns cannot be specified.

There are two different approaches we could use to accomplish table partitioning. The first is to create a new partitioned table and then simply copy the data from your existing table into the new table and do a table rename. The second approach is to partition an existing table by rebuilding or creating a clustered index on the table.

An example of horizontal partitioning with creating a new partitioned table

SQL Server 2005 introduced a built-in partitioning feature to horizontally partition a table with up to 1000 partitions in SQL Server 2008, and 15000 partitions in SQL Server 2012, and the data placement is handled automatically by SQL Server. This feature is available only in the Enterprise Edition of SQL Server.

To create a partitioned table for storing monthly reports we will first create additional filegroups. A filegroup is a logical storage unit. Every database has a primary filegroup that contains the primary data file (.mdf). An additional, user-defined, filegrups can be created to contain secondary files (.ndf). We will create 12 filegroups for every month:

|  |
| --- |
| ALTER DATABASE PartitioningDB  ADD FILEGROUP January  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP February  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP March  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP April  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP May  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP June  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP July  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP Avgust  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP September  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP October  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP November  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP December  GO |

To check created and available file groups in the current database run the following query:

|  |
| --- |
| SELECT name AS AvailableFilegroups  FROM sys.filegroups  WHERE type = 'FG' |



When filegrups are created we will add .ndf file to every filegroup:

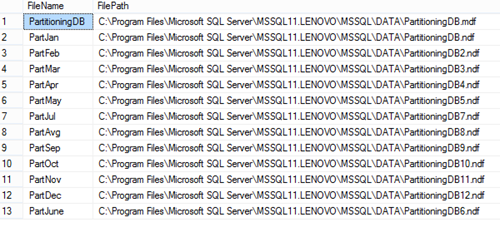
|  |
| --- |
| ALTER DATABASE [PartitioningDB]      ADD FILE      (      NAME = [PartJan],      FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.LENOVO\MSSQL\DATA\PartitioningDB.ndf',          SIZE = 3072 KB,          MAXSIZE = UNLIMITED,          FILEGROWTH = 1024 KB      ) TO FILEGROUP [January] |

The same way files to all created filegroups with specifying the logical name of the file and the operating system (physical) file name for each filegroup e.g.:

|  |
| --- |
| ALTER DATABASE [PartitioningDB]      ADD FILE      (      NAME = [PartFeb],      FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.LENOVO\MSSQL\DATA\PartitioningDB2.ndf',          SIZE = 3072 KB,          MAXSIZE = UNLIMITED,          FILEGROWTH = 1024 KB      ) TO FILEGROUP [February] |

To check files created added to the filegroups run the following query:

|  |
| --- |
| SELECT  name as [FileName],  physical\_name as [FilePath]  FROM sys.database\_files  where type\_desc = 'ROWS'  GO |

[](https://www.sqlshack.com/wp-content/uploads/2014/04/PartitioningDB.png)

After creating additional filegroups for storing data we’ll create a partition function. A partition function is a function that maps the rows of a partitioned table into partitions based on the values of a partitioning column. In this example we will create a partitioning function that partitions a table into 12 partitions, one for each month of a year’s worth of values in a datetime column:

|  |
| --- |
| CREATE PARTITION FUNCTION [PartitioningByMonth] (datetime)  AS RANGE RIGHT FOR VALUES ('20140201', '20140301', '20140401',                 '20140501', '20140601', '20140701', '20140801',                 '20140901', '20141001', '20141101', '20141201'); |

To map the partitions of a partitioned table to filegroups and determine the number and domain of the partitions of a partitioned table we will create a partition scheme:

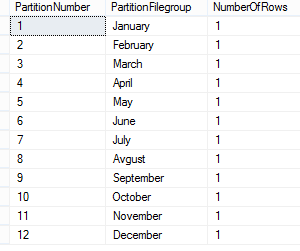
|  |
| --- |
| CREATE PARTITION SCHEME PartitionBymonth  AS PARTITION PartitioningBymonth  TO (January, February, March,      April, May, June, July,      Avgust, September, October,      November, December); |

Now we’re going to create the table using the PartitionBymonth partition scheme, and fill it with the test data:

|  |
| --- |
| CREATE TABLE Reports  (ReportDate datetime PRIMARY KEY,  MonthlyReport varchar(max))  ON PartitionBymonth (ReportDate);  GO    INSERT INTO Reports (ReportDate,MonthlyReport)  SELECT '20140105', 'ReportJanuary' UNION ALL  SELECT '20140205', 'ReportFebryary' UNION ALL  SELECT '20140308', 'ReportMarch' UNION ALL  SELECT '20140409', 'ReportApril' UNION ALL  SELECT '20140509', 'ReportMay' UNION ALL  SELECT '20140609', 'ReportJune' UNION ALL  SELECT '20140709', 'ReportJuly' UNION ALL  SELECT '20140809', 'ReportAugust' UNION ALL  SELECT '20140909', 'ReportSeptember' UNION ALL  SELECT '20141009', 'ReportOctober' UNION ALL  SELECT '20141109', 'ReportNovember' UNION ALL  SELECT '20141209', 'ReportDecember' |

We will now verify the rows in the different partitions:

|  |
| --- |
| SELECT  p.partition\_number AS PartitionNumber,  f.name AS PartitionFilegroup,  p.rows AS NumberOfRows  FROM sys.partitions p  JOIN sys.destination\_data\_spaces dds ON p.partition\_number = dds.destination\_id  JOIN sys.filegroups f ON dds.data\_space\_id = f.data\_space\_id  WHERE OBJECT\_NAME(OBJECT\_ID) = 'Reports' |

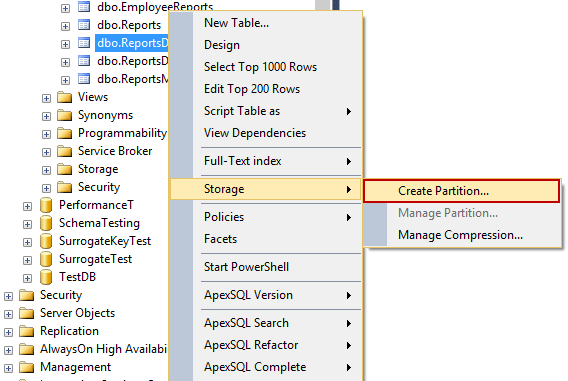


Now just copy data from your table and rename a partitioned table.

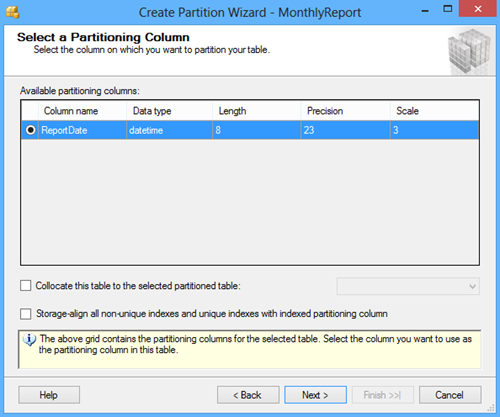
Partitioning a table using the SQL Server Management Studio Partitioning wizard

SQL Server 2008 introduced a table partitioning wizard in SQL Server Management Studio.

Right click on a table in the **Object Explorer** pane and in the **Storage**context menu choose the **Create Partition** command:

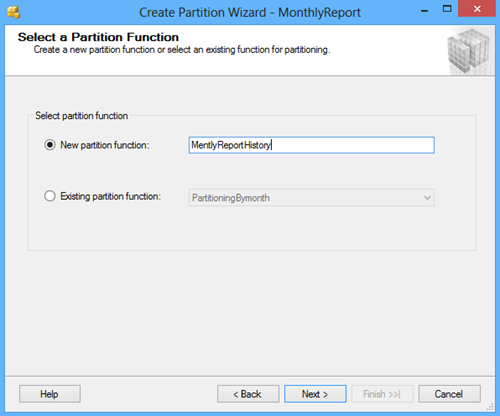


In the **Select a Partitioning Column** window, select a column which will be used to partition a table from available partitioning columns:

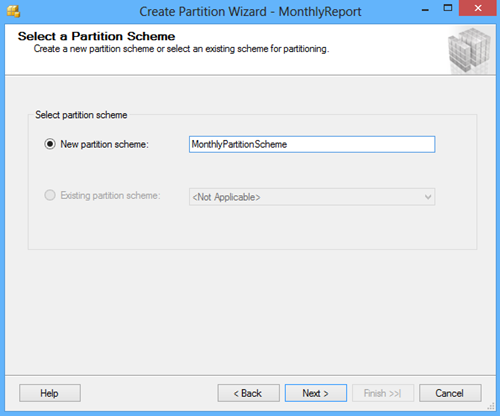
[](https://www.sqlshack.com/wp-content/uploads/2014/04/SelectAPartitioningColumn.png)

Other options in the **Create Partition Wizard** dialog include the **Collocate this table to the selected partition table** option used to display related data to join with the partitioned column and the **Storage Align Non Unique Indexes and Unique Indexes with an Indexed Partition Column** option that aligns all indexes of the partitioned table with the same partition scheme.

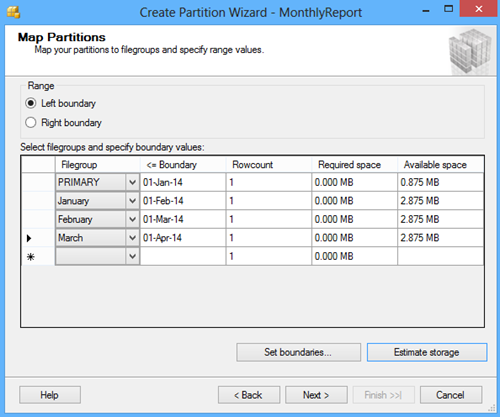
After selecting a column for partitioning click the Next button. In the **Select a Partition Function** window enter the name of a partition function to map the rows of the table or index into partitions based on the values of the ReportDate column, or choose the existing partition function:

[](https://www.sqlshack.com/wp-content/uploads/2014/04/SelectAPartitioningFunction.png)

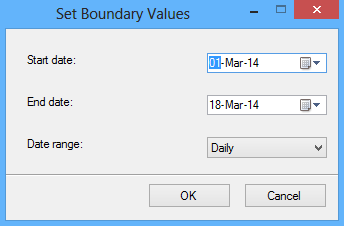
Click the Next button and in the **Select a Partition Scheme** window create the partition scheme to map the partitions of the MonthlyReport table to different filegroups:

[](https://www.sqlshack.com/wp-content/uploads/2014/04/SelectAPartitioningScheme.png)

Click the Next button and in the **Map Partitions** window choose the rage of partitioning and select the available filegroups and the range boundary. The Left boundary is based on Value <= Boundary and the Right boundary is based on Value < Boundary.

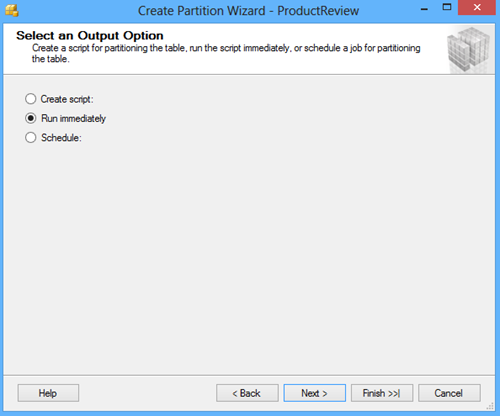
[](https://www.sqlshack.com/wp-content/uploads/2014/04/MapPartitions.png)

By clicking the **Set boundaries** button you can customize the date range and set the start and the end date for each partition:

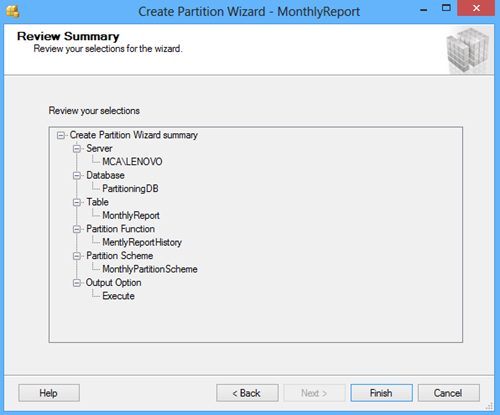


The **Estimate storage** option determines the Rowcount, the Required space, and the Available space columns that displays an estimate on required space and available space based on number of records in the table.

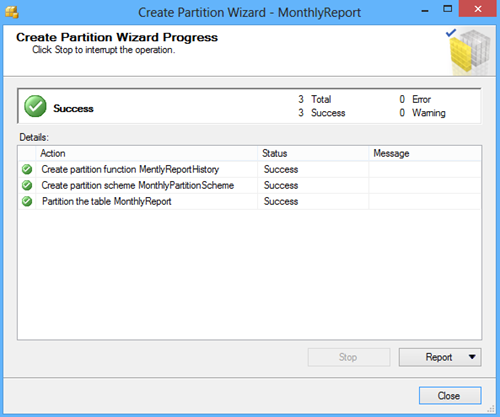
The next screen of the wizard offers to choose the option to whether to execute the script immediately by the wizard to create objects and a partition table, or to create a script and save it. A schedule for executing the script to perform the operations automatically can also be specified:

[](https://www.sqlshack.com/wp-content/uploads/2014/04/SelectAnOutputOption.png)

The next screen of the wizard shows a review of selections made in the wizard:

[](https://www.sqlshack.com/wp-content/uploads/2014/04/ReviewSummary.png)

Click the Finish button to complete the process:

[](https://www.sqlshack.com/wp-content/uploads/2014/04/CreatePartitionWizardProgress.png)

There are many benefits of partitioning large tables. You can speed up loading and archiving of data, you can perform maintenance operations on individual partitions instead of the whole table, and you may be able to improve query performance. However, implementing table partitioning is not a trivial task and you need a good understanding of how it works to implement and use it correctly.

Being a business intelligence and data warehouse developer, not a DBA, it took me a while to understand table partitioning. I had to read a lot, get plenty of hands-on experience and make some mistakes along the way. (The illustration to the left is my Table Partitioning Cheat Sheet.) One of my favorite ways to learn something is to figure out how to explain it to others, so I recently did a webinar about table partitioning. I wanted to follow that up with focused blog posts that included answers to questions I received during the webinar. This post covers the basics of partitioned tables, partition columns, partition functions and partition schemes.

## *What is Table Partitioning?*

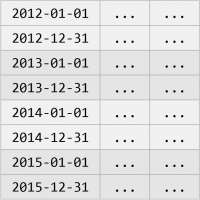
[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionedTable.png)

Table partitioning is a way to divide a large table into smaller, more manageable parts without having to create separate tables for each part. Data in a partitioned table is physically stored in groups of rows called partitions and each partition can be accessed and maintained separately. Partitioning is not visible to end users, a partitioned table behaves like one logical table when queried.

This example illustration is used throughout this blog post to explain basic concepts. The table contains data from every day in 2012, 2013, 2014 and 2015, and there is one partition per year. To simplify the example, only the first and last day in each year is shown.

An alternative to partitioned tables (for those who don’t have Enterprise Edition) is to create separate tables for each group of rows, union the tables in a view and then query the view instead of the tables. This is called a partitioned view. (Partitioned views are not covered in this blog post.)

## *What is a Partition Column?*

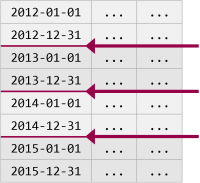
[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionKey.png)

Data in a partitioned table is partitioned based on a single column, the partition column, often called the partition key. Only one column can be used as the partition column, but it is possible to use a computed column.

In the example illustration the date column is used as the partition column. SQL Server places rows in the correct partition based on the values in the date column. All rows with dates before or in 2012 are placed in the first partition, all rows with dates in 2013 are placed in the second partition, all rows with dates in 2014 are placed in the third partition, and all rows with dates in 2015 or after are placed in the fourth partition. If the partition column value is NULL, the rows are placed in the first partition.

It is important to select a partition column that is almost always used as a filter in queries. When the partition column is used as a filter in queries, SQL Server can access only the relevant partitions. This is called partition elimination and can greatly improve performance when querying large tables.

## *What is a Partition Function?*

[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction.png)

The partition function defines how to partition data based on the partition column. The partition function does not explicitly define the partitions and which rows are placed in each partition. Instead, the partition function specifies boundary values, the points between partitions. The total number of partitions is always the total number of boundary values + 1.

In the example illustration there are three boundary values. The first boundary value is between 2012 and 2013, the second boundary value is between 2013 and 2014, and the third boundary value is between 2014 and 2015. The three boundary values create four partitions. (The first partition also includes all rows with dates before 2012 and the last partition also includes all rows after 2015, but the example is kept simple with only four years for now.)

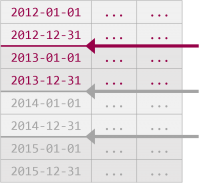
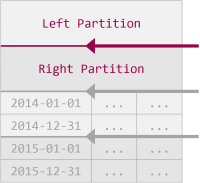
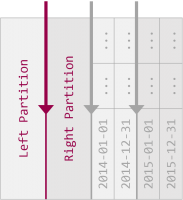
But what are the actual boundary values used in the example? How do you know which date values are the points between two years? Is it December 31st or January 1st? The answer is that it can actually be either December 31st or January 1st, it depends on whether you use a range left or a range right partition function.

### *Range Left and Range Right*

Partition functions are created as either range left or range right to specify whether the boundary values belong to their left or right partitions:

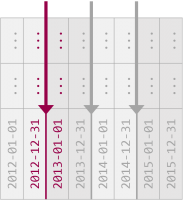
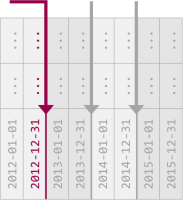
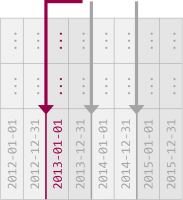
* Range left means that the actual boundary value belongs to its left partition, it is the last value in the left partition.
* Range right means that the actual boundary value belongs to its right partition, it is the first value in the right partition.

Left and right partitions make more sense if the table is rotated:

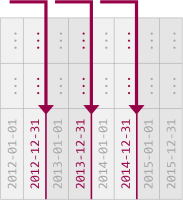
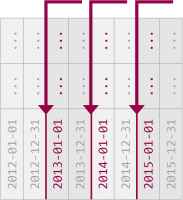
[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction01.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction02.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction03.png)

### *Range Left and Range Right using Dates*

The first boundary value is between 2012 and 2013. This can be created in two ways, either by specifying a range left partition function with December 31st as the boundary value, or as a range right partition function with January 1st as the boundary value:

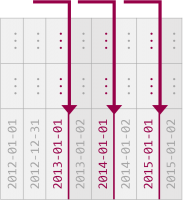
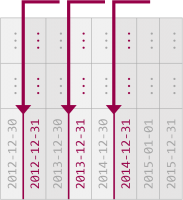
[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction04.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction05.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction06.png)

Partition functions are created as either range left or range right, it is not possible to combine both in the same partition function. In a range left partition function, all boundary values are upper boundaries, they are the last values in the partitions. If you partition by year, you use December 31st. If you partition by month, you use January 31st, February 28th / 29th, March 31st, April 30th and so on. In a range right partition function, all boundary values are lower boundaries, they are the first values in the partitions. If you partition by year, you use January 1st. If you partition by month, you use January 1st, February 1st, March 1st, April 1st and so on:

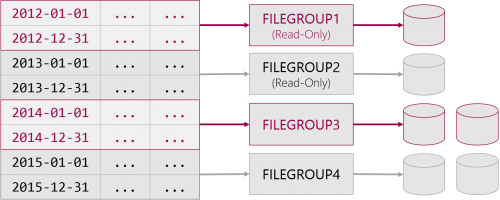
[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction07.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction08.png)

### *Range Left and Range Right using the*Wrong Dates

If the wrong dates are used as boundary values, the partitions incorrectly span two time periods:

[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction10.png) → [](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionFunction09.png)

## *What is a Partition Scheme?*

[](https://www.cathrinewilhelmsen.net/scribbles/wp-content/uploads/2015/04/PartitionScheme.png)

The partition scheme maps the logical partitions to physical filegroups. It is possible to map each partition to its own filegroup or all partitions to one filegroup.

A filegroup contains one or more data files that can be spread on one or more disks. Filegroups can be set to read-only, and filegroups can be backed up and restored individually. There are many benefits of mapping each partition to its own filegroup. Less frequently accessed data can be placed on slower disks and more frequently accessed data can be placed on faster disks. Historical, unchanging data can be set to read-only and then be excluded from regular backups. If data needs to be restored it is possible to restore the partitions with the most critical data first.

## *How do I create a Partitioned Table?*

The following script (for SQL Server 2012 and higher) first creates a numbers table function created by Itzik Ben-Gan that is used to insert test data. The script then creates a partition function, a partition scheme and a partitioned table. (It is important to notice that this script is meant to demonstrate the basic concepts of table partitioning, it does not create any indexes or constraints and it maps all partitions to the [PRIMARY] filegroup. This script is not meant to be used in a real-world project.) Finally it inserts test data and shows information about the partitioned table.

*/\* --------------------------------------------------*

*-- Create helper function GetNums by Itzik Ben-Gan*

*-- https://www.itprotoday.com/sql-server/virtual-auxiliary-table-numbers*

*-- GetNums is used to insert test data*

*-------------------------------------------------- \*/*

*-- Drop helper function if it already exists*

IF OBJECT\_ID('GetNums') IS NOT NULL

DROP FUNCTION GetNums;

GO

*-- Create helper function*

CREATE FUNCTION GetNums(@n AS BIGINT) RETURNS TABLE AS RETURN

WITH

L0 AS(SELECT 1 AS c UNION ALL SELECT 1),

L1 AS(SELECT 1 AS c FROM L0 AS A CROSS JOIN L0 AS B),

L2 AS(SELECT 1 AS c FROM L1 AS A CROSS JOIN L1 AS B),

L3 AS(SELECT 1 AS c FROM L2 AS A CROSS JOIN L2 AS B),

L4 AS(SELECT 1 AS c FROM L3 AS A CROSS JOIN L3 AS B),

L5 AS(SELECT 1 AS c FROM L4 AS A CROSS JOIN L4 AS B),

Nums AS(SELECT ROW\_NUMBER() OVER(ORDER BY (SELECT NULL)) AS n FROM L5)

SELECT TOP (@n) n FROM Nums ORDER BY n;

GO

*/\* ------------------------------------------------------------*

*-- Create example Partitioned Table (Heap)*

*-- The Partition Column is a DATE column*

*-- The Partition Function is RANGE RIGHT*

*-- The Partition Scheme maps all partitions to [PRIMARY]*

*------------------------------------------------------------ \*/*

*-- Drop objects if they already exist*

IF EXISTS (SELECT \* FROM sys.tables WHERE name = N'Sales')

DROP TABLE Sales;

IF EXISTS (SELECT \* FROM sys.partition\_schemes WHERE name = N'psSales')

DROP PARTITION SCHEME psSales;

IF EXISTS (SELECT \* FROM sys.partition\_functions WHERE name = N'pfSales')

DROP PARTITION FUNCTION pfSales;

*-- Create the Partition Function*

CREATE PARTITION FUNCTION pfSales (DATE)

AS RANGE RIGHT FOR VALUES

('2013-01-01', '2014-01-01', '2015-01-01');

*-- Create the Partition Scheme*

CREATE PARTITION SCHEME psSales

AS PARTITION pfSales

ALL TO ([Primary]);

*-- Create the Partitioned Table (Heap) on the Partition Scheme*

CREATE TABLE Sales (

SalesDate DATE,

Quantity INT

) ON psSales(SalesDate);

*-- Insert test data*

INSERT INTO Sales(SalesDate, Quantity)

SELECT DATEADD(DAY,dates.n-1,'2012-01-01') AS SalesDate, qty.n AS Quantity

FROM GetNums(DATEDIFF(DD,'2012-01-01','2016-01-01')) dates

CROSS JOIN GetNums(1000) AS qty;

*-- View Partitioned Table information*

SELECT

OBJECT\_SCHEMA\_NAME(pstats.object\_id) AS SchemaName

,OBJECT\_NAME(pstats.object\_id) AS TableName

,ps.name AS PartitionSchemeName

,ds.name AS PartitionFilegroupName

,pf.name AS PartitionFunctionName

,CASE pf.boundary\_value\_on\_right WHEN 0 THEN 'Range Left' ELSE 'Range Right' END AS PartitionFunctionRange

,CASE pf.boundary\_value\_on\_right WHEN 0 THEN 'Upper Boundary' ELSE 'Lower Boundary' END AS PartitionBoundary

,prv.value AS PartitionBoundaryValue

,c.name AS PartitionKey

,CASE

WHEN pf.boundary\_value\_on\_right = 0

THEN c.name + ' > ' + CAST(ISNULL(LAG(prv.value) OVER(PARTITION BY pstats.object\_id ORDER BY pstats.object\_id, pstats.partition\_number), 'Infinity') AS VARCHAR(100)) + ' and ' + c.name + ' <= ' + CAST(ISNULL(prv.value, 'Infinity') AS VARCHAR(100))

ELSE c.name + ' >= ' + CAST(ISNULL(prv.value, 'Infinity') AS VARCHAR(100)) + ' and ' + c.name + ' < ' + CAST(ISNULL(LEAD(prv.value) OVER(PARTITION BY pstats.object\_id ORDER BY pstats.object\_id, pstats.partition\_number), 'Infinity') AS VARCHAR(100))

END AS PartitionRange

,pstats.partition\_number AS PartitionNumber

,pstats.row\_count AS PartitionRowCount

,p.data\_compression\_desc AS DataCompression

FROM sys.dm\_db\_partition\_stats AS pstats

INNER JOIN sys.partitions AS p ON pstats.partition\_id = p.partition\_id

INNER JOIN sys.destination\_data\_spaces AS dds ON pstats.partition\_number = dds.destination\_id

INNER JOIN sys.data\_spaces AS ds ON dds.data\_space\_id = ds.data\_space\_id

INNER JOIN sys.partition\_schemes AS ps ON dds.partition\_scheme\_id = ps.data\_space\_id

INNER JOIN sys.partition\_functions AS pf ON ps.function\_id = pf.function\_id

INNER JOIN sys.indexes AS i ON pstats.object\_id = i.object\_id AND pstats.index\_id = i.index\_id AND dds.partition\_scheme\_id = i.data\_space\_id AND i.type <= 1 */\* Heap or Clustered Index \*/*

INNER JOIN sys.index\_columns AS ic ON i.index\_id = ic.index\_id AND i.object\_id = ic.object\_id AND ic.partition\_ordinal > 0

INNER JOIN sys.columns AS c ON pstats.object\_id = c.object\_id AND ic.column\_id = c.column\_id

LEFT JOIN sys.partition\_range\_values AS prv ON pf.function\_id = prv.function\_id AND pstats.partition\_number = (CASE pf.boundary\_value\_on\_right WHEN 0 THEN prv.boundary\_id ELSE (prv.boundary\_id+1) END)

WHERE pstats.object\_id = OBJECT\_ID('Sales')

ORDER BY TableName, PartitionNumber;