SQL Server Partition

- A useful strategy in large databases is to partition a single logical set of data into multiple physical storage locations for **manageability** or **performance** reasons.
- **SQL Server 2005** supports partitioned tables and indexes.
- Partitioned tables and indexes are single objects stored in multiple physical partitions, each partition holding a specific subset of the data.
- Why Partition?
- What Are Partition Functions?
- What Is a Partition Scheme?
- What Are Partitioned Tables?
- What Operations Can Be Performed on Partitioned Data?
- Demo: Creating a Partitioned Table
- Lab: Creating a Partitioned Table

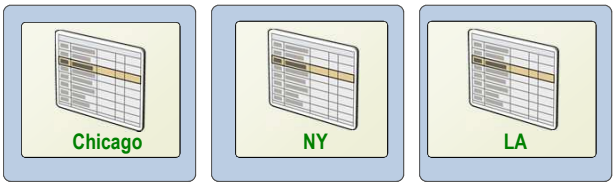
Table Partitioning Overview

- **Horizontal partitioning:**
  - Where selected subsets of rows are placed in different tables.
  - When a view is created over all the tables, and queries directed to the view, the result is a partitioned view.
  - In a **partitioned view**, you have to manually apply the required constraints and operations, and maintenance can be complex and time-consuming.
- **Vertical partitioning:**
  - Where the columns of a very wide table are spread across multiple tables containing distinct subsets of the columns with the same number of rows.
  - The result is multiple tables containing the same number of rows but different columns, usually with the same primary key column in each table.
  - A view is defined across the multiple tables and queries directed against the view.
  - SQL Server does not provide built-in support for vertical partitioning.
- **SQL Server's table partitioning:**
  - Differs from the above two approaches by partitioning a single table.
  - Multiple physical tables are no longer involved.
  - When a table is created as a partitioned table, SQL Server automatically places the table's rows in the correct partition, and SQL Server maintains the partitions behind the scenes.
  - You can then perform maintenance operations on individual partitions, and properly filtered queries will access only the correct partitions.
  - But it is still **one table** as far as SQL Server is concerned.

Partitioning Types

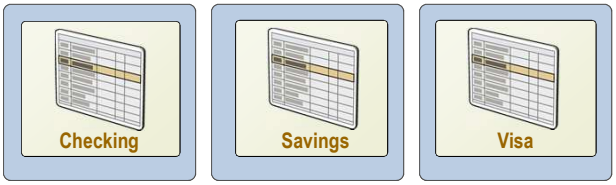
Horizontal partitioning:

Unique Regional ID

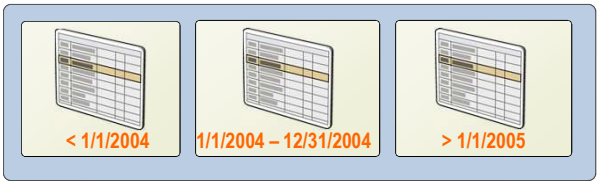


Vertical partitioning:

Customer SSN



SQL Server's table partitioning:



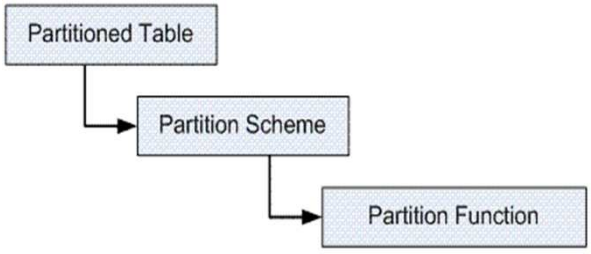
Why Partition?

- **The concept of "Divide and Conquer".**
  - "Dividing Tables and Indexes into manageable pieces."
  - The Best Thing Since ***Sliced Bread***
- **Benefits**
  - Each partition is stored in its own physical location and can be managed individually.
  - It can function independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.
  - Operations on partitioned tables and indexes are performed in parallel by assigning different parallel execution servers.
  - It's entirely transparent to SQL statements, partitioning can be applied to almost any application.
- **Storing partitions in separate location enables you to:**
  - Reduce the possibility of data corruption in multiple partitions
  - Control the mapping of partitions to disk drives (important for balancing I/O load)
  - Improve manageability, availability, and performance
- **Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data.**
  - OLTP systems often benefit from improvements in manageability and availability.
  - Data warehousing systems benefit from performance and manageability

Planning for Table Partitioning

In order to successfully partition a large table, you must make a Number of decisions. In particular, you need to:

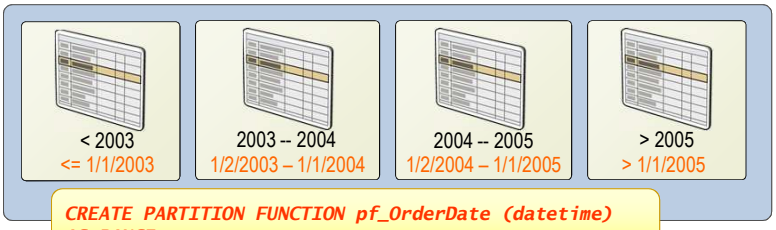
- Plan the partitioning:
  - Decide which table or tables can benefit from the increased manageability and availability of partitioning.
  - Decide the column or column combination upon which to base the partition.
- Specify the partition boundaries in a partition function.
- Plan on how to store the partitions in filegroups using a partition scheme.



Bottom up step-by-step implementation process

What Are Partition Functions?

- A partition function specifies the data type of the key used to partition the data and the boundary values for each partition.
- The number of partitions defined by a partition function is always one more than the number of boundary values that the function defines.
  - Ex: a partition function that defines a **datetime** partitioning key with the **boundary values** '01/01/2003', '01/01/2004', and '01/01/2005' will result in four partitions:
    - one for everything up to the first boundary value - values before January 2, 2003
    - one for values between the first and second boundaries - values between January 1, 2003, and January 2, 2004
    - one for values between the second and third boundaries - values between January 1, 2004, and January 2, 2005
    - one for everything beyond the final boundary - values after January 1, 2005




```
CREATE PARTITION FUNCTION pf_OrderDate (datetime)
AS RANGE
FOR VALUES ('01/01/2003', '01/01/2004', '01/01/2005')
```

Create Partition Function Syntax

```
CREATE PARTITION FUNCTION partition_function_name ( input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] )
[ ; ]
```

- ***partition_function_name:*** the name of the partition function. Partition function names must be unique within the database and comply with the rules for identifiers.
- ***input_parameter_type:*** the data type of the column used for partitioning. All data types are valid for use as partitioning columns, except text, ntext, image, xml, timestamp, varchar(max), nvarchar(max), varbinary(max), alias data types, or CLR user-defined data types.
  - The actual column, known as a partitioning column, is specified in the CREATE TABLE or CREATE INDEX statement.
- ***boundary_value*:** Specifies the boundary values for each partition of a partitioned table or index that uses *partition_function_name*.
  - If *boundary_value* is empty, the partition function maps the whole table or index into a single partition.
  - Can only one partitioning column, specified in a CREATE TABLE or CREATE INDEX statement.
- ***...n:*** Specifies the number of values supplied by *boundary_value*, **not to exceed 999**.
  - The number of partitions created is equal to ***n* + 1**.
  - The values do not have to be listed in order. If values are not in order, SQL Server sorts them, creates the function, and returns a warning that the values are not provided in order.
- **LEFT | RIGHT:** Partition functions are configured as LEFT or RIGHT, depending on where you want data that matches the boundary value to go.
  - LEFT partition: exact matches with the boundary value go to the partition to the left. LEFT is the default.
  - RIGHT partition: exact matches with the boundary value go to the partition to the right.

Create Partition Function Example – 1 & 2

1. Creating a RANGE LEFT partition function on an int column

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
```

- The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	col1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1000	col1 > 1000

2. Creating a RANGE RIGHT partition function on an int column

```
CREATE PARTITION FUNCTION myRangePF2 (int)
AS RANGE RIGHT FOR VALUES (1, 100, 1000);
```

- The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	col1 < 1	col1 >= 1 AND col1 < 100	col1 >= 100 AND col1 < 1000	col1 >= 1000

Create Partition Function Examples – 3 & 4

3. Creating a RANGE LEFT partition function on a char column

```
CREATE PARTITION FUNCTION myRangePF3 (char(20))
AS RANGE LEFT FOR VALUES ('EX', 'RXE', 'XR');
```

- The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	col1 <= EX...	col1 > EX AND col1 <= RXE...	col1 > RXE AND col1 <= XR...	col1 > XR

4. Creating a RANGE RIGHT partition function on a char column

```
CREATE PARTITION FUNCTION myRangePF3 (char(20))
AS RANGE RIGHT FOR VALUES ('EX', 'RXE', 'XR');
```

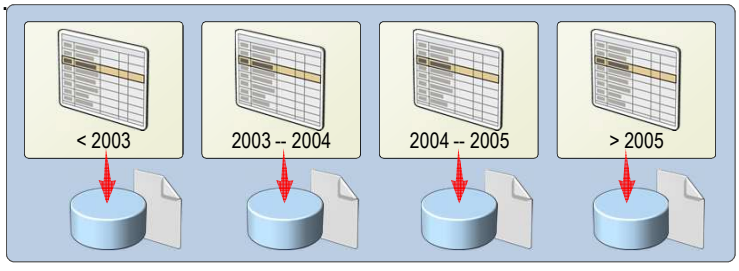
- The following table shows how a table that uses this partition function on partitioning column **col1** would be partitioned.

Partition	1	2	3	4
Values	col1 < EX...	col1 >= EX AND col1 < RXE...	col1 >= RXE AND col1 < XR...	col1 >= XR

Page 9

What Is a Partition Scheme?

- A **partition function** maps the rows of a table or index into partitions based on the values of a specified column.
- A **partition scheme** maps the partitions defined in a partition function to the filegroups on which the partitions will be physically stored.
- You can map all partitions to the same filegroup, or you can map some or all of the partitions to different filegroups, depending on your specific needs.
- **Next filegroup:** When you create a partition scheme, you can optionally specify the filegroup to be used if another partition is added to the partition function.




```
CREATE PARTITION SCHEME ps_OrderDate
AS PARTITION pf_OrderDate
TO (fg1, fg2, fg3, fg4, fg5)
```

Page 10

Create Partition Scheme Syntax

```
CREATE PARTITION SCHEME partition_scheme_name
AS PARTITION partition_function_name
[ ALL ] TO ( { file_group_name | [ PRIMARY ] } [ ,...n ] )
[ ; ]
```

- ***partition_scheme_name:*** The name of the partition scheme. Partition scheme names must be unique within the database and comply with the rules for identifiers.
- ***partition_function_name:*** The name of the partition function using the partition scheme.
  - Partitions created by the partition function are mapped to the filegroups specified in the partition scheme.
- **ALL:** Specifies that all partitions map to the filegroup provided in *file_group_name*, or to the primary filegroup if **[**PRIMARY**]** is specified.
  - If ALL is specified, only one *file_group_name* can be specified.
- ***file_group_name* | [PRIMARY] [,...*n*]:** Specifies the names of the filegroups to hold the partitions specified by *partition_function_name*.
  - If **[**PRIMARY**]** is specified, the partition is stored on the primary filegroup.
  - If ALL is specified, only one *file_group_name* can be specified.
  - Partitions are assigned to filegroups, starting with partition 1, in the order in which the filegroups are listed in [,...*n*].
  - The same *file_group_name* can be specified more than one time in [,...*n*].
  - If *partition_function_name* generates less partitions than filegroups, the first unassigned filegroup is marked NEXT USED, and an information message displays naming the NEXT USED filegroup.
  - When you specify the primary filegroup in *file_group_name* [1,...*n*], PRIMARY must be delimited, as in **[**PRIMARY**]**, because it is a keyword.

```
CREATE PARTITION SCHEME ps_OrderDate
AS PARTITION pf_OrderDate
ALL TO (PRIMARY))
```


Create Partition Scheme Examples – 1 & 2

1. Creating a partition scheme that maps each partition to a different filegroup

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS2
AS PARTITION myRangePF1 TO ( test1fg, test2fg, test3fg, test4fg );
```

- A partition scheme is then created that specifies the filegroups to hold each one of the four partitions.

Filegroup	test1fg	test2fg	test3fg	test4fg
Partition	1	2	3	4
Values	col1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1000	col1 > 1000

2. Creating a partition scheme that maps multiple partitions to the same filegroup

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS2
AS PARTITION myRangePF1 TO ( test1fg, test1fg, test1fg, test2fg );
```

- The same filegroup is repeated for different partitions.

Filegroup	test1fg	test1fg	test1fg	test2fg
Partition	1	2	3	4
Values	col1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1000	col1 > 1000

Create Partition Scheme Examples – 3 & 4

1. Creating a partition scheme maps all partitions to the same filegroup.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS3
AS PARTITION myRangePF1 ALL TO ( test1fg );
```

- Only one filegroup for all the partitions.

Filegroup	test1fg	test1fg	test1fg	test1fg
Partition	1	2	3	4
Values	col1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1000	col1 > 1000

2. Creating a partition scheme that specifies a 'NEXT USED' filegroup

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS4
AS PARTITION myRangePF1 TO (test1fg, test2fg, test3fg, test4fg, test5fg)
```

Partition scheme 'myRangePS4' has been created successfully.
'test5fg' is marked as the next used filegroup in partition scheme 'myRangePS4'.

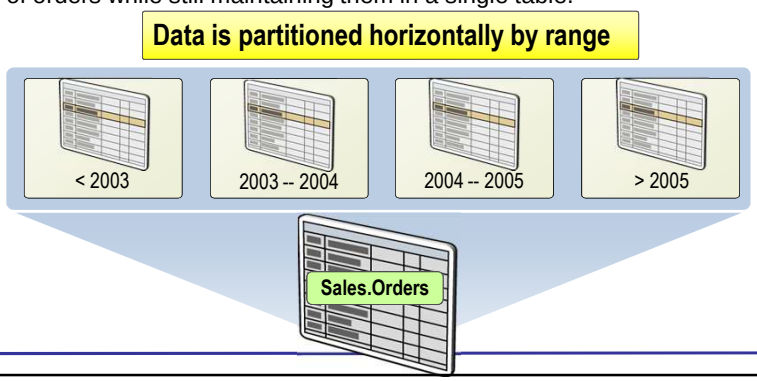
- a partition scheme is created that lists more filegroups than there are partitions created by the associated partition function.

Filegroup	test1fg	test2fg	test3fg	test4fg	test5fg
Partition	1	2	3	4	5
Values	col1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1000	col1 > 1000	

13

What Are Partitioned Tables?

- A partitioned table is a table in which the data is separated horizontally into multiple physical locations based on a range of values for a specific column.
- The physical locations for partitions are filegroups.
  - Ex: Use a partitioned table to store sales orders and then separate the order records into different filegroups based on the order date so that orders placed in the current financial year are stored in one partition.
  - Orders placed in the previous financial year are stored in a second partition
  - All orders older than two years are stored in a third partition.
  - This technique makes it possible to control the physical storage of different kinds of orders while still maintaining them in a single table.



Page 14

Manageability benefits of partitioned tables

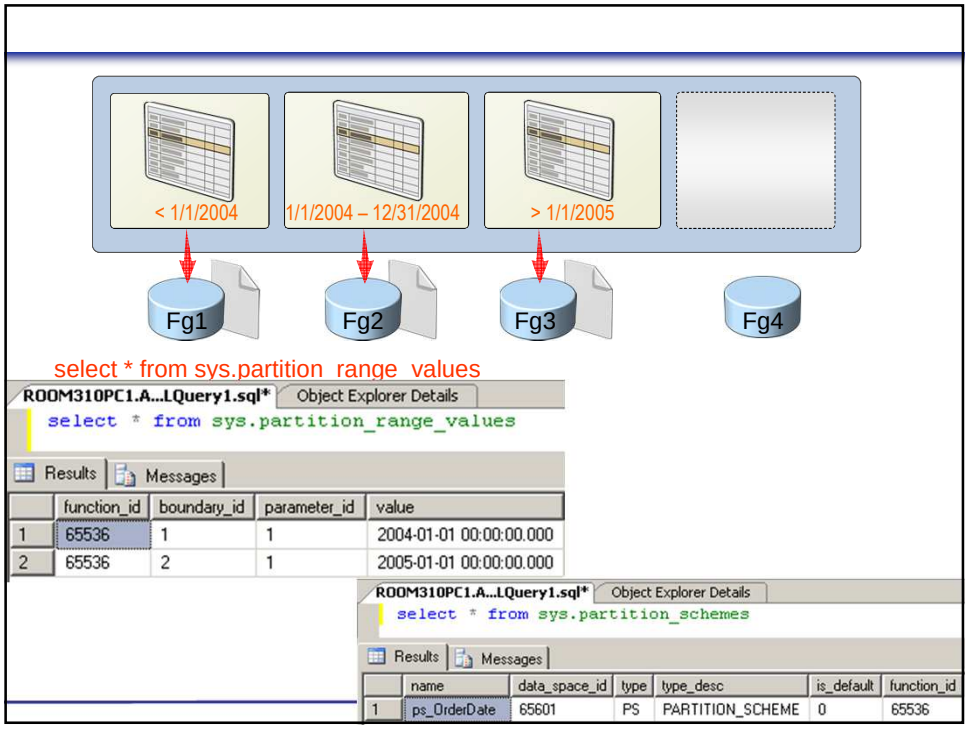
The main reason for implementing a partitioned table is to make it easier to manage different sets of data within the same table. Manageability benefits of partitioned tables:

- ***The ability to implement separate backup strategies:*** Different sets of data might have different backup requirements. For example, recent orders data might be updated frequently and require regular backups, while older orders might change rarely and require only infrequent backups.
- ***Control over storage media:*** Partitioning a table lets you choose appropriate storage for data, based on its access requirements. For example, you can store historical, unchanging data on an NTFS compressed filegroup while keeping the current data on a high-performance redundant array of independent disks (RAID 1+0) filegroup.
- ***Index management benefits:*** In addition to partitioning a table, you can partition its indexes. This allows you to reorganize, optimize, and rebuild indexes by partition, which is faster and less intrusive than managing an entire index. Additionally, partitioning an index can minimize fragmentation. For example, older order data is unlikely to change, so the index pages that relate to historical orders are stable.
- ***Performance benefits:***
  - ***Faster index searches*** - Partitioning results in smaller index trees for each partition, making partition access fast, especially when you limit rows by specifying the partition key in a WHERE clause.
  - ***Enhanced JOIN performance*** - JOIN operations can be faster when joining aligned tables (that is, tables that are partitioned in the same manner).
  - ***Reduced locking*** - Lock escalation stops at the partition level, significantly minimizing the risk of blocking and deadlock.

Demo: Creating A Partitioned Table

```
-- 1. Create partition function
USE AdventureWorks
GO
CREATE PARTITION FUNCTION pf_OrderDate (datetime)
AS RANGE RIGHT
FOR VALUES ('01/01/2004', '01/01/2005')
```

```
-- 2. Add filegroups
ALTER DATABASE AdventureWorks ADD FILEGROUP fg1
ALTER DATABASE AdventureWorks ADD FILEGROUP fg2
ALTER DATABASE AdventureWorks ADD FILEGROUP fg3
ALTER DATABASE AdventureWorks ADD FILEGROUP fg4
GO
ALTER DATABASE AdventureWorks
ADD FILE
( NAME = data1,
  FILENAME = 'c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\AWd1.ndf',
  SIZE = 1MB,
  MAXSIZE = 2MB,
  FILEGROWTH = 1MB)
TO FILEGROUP fg1
        :
```



select * from sys.partition_range_values

```
select * from sys.partition_range_values
```

	function_id	boundary_id	parameter_id	value
1	65536	1	1	2004-01-01 00:00:00.000
2	65536	2	1	2005-01-01 00:00:00.000

```
select * from sys.partition_schemes
```

	name	data_space_id	type	type_desc	is_default	function_id
1	ps_OrderDate	65601	PS	PARTITION_SCHEME	0	65536

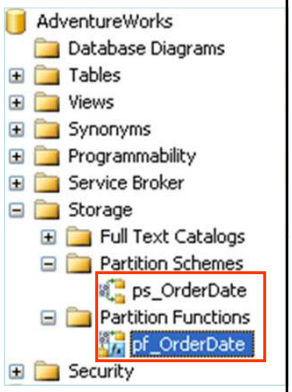
```
-- 3. Create partition scheme
CREATE PARTITION SCHEME ps_OrderDate
AS PARTITION pf_OrderDate
TO (fg1, fg2, fg3, fg4)
```

Partition scheme 'ps_OrderDate' has been created successfully.
'fg4' is marked as the next used filegroup in partition scheme 'ps_OrderDate'.

Select * from sys.partition_schemes

```
select * from sys.partition_schemes
```

	name	data_space_id	type	type_desc	is_default	function_id
1	ps_OrderDate	65601	PS	PARTITION_SCHEME	0	65536




```
-- 4. Create partitioned table
CREATE TABLE dbo.PartitionedTransactions
    (  TransactionID int IDENTITY(1,1) NOT NULL,
       ProductID int NOT NULL,
       TransactionDate datetime NOT NULL DEFAULT (getdate()),
       TransactionType nchar(1) NOT NULL)
ON ps_OrderDate(TransactionDate)
```

```
-- 5. Insert data
INSERT INTO dbo.PartitionedTransactions
SELECT ProductID, TransactionDate, TransactionType
FROM Production.TransactionHistory
INSERT INTO dbo.PartitionedTransactions
VALUES (1, '01/01/2005', 'S')
```

(113443 row(s) affected)
(1 row(s) affected)

Page 18

```
-- 6. View partition metadata
SELECT * FROM sys.Partitions
WHERE [object_id] = OBJECT_ID('dbo.PartitionedTransactions')
```

Results | Messages

	partition_id	object_id	index_id	partition_number	hobt_id	rows
1	72057594054508544	791673868	0	1	72057594054508544	42844
2	72057594054574080	791673868	0	2	72057594054574080	70599
3	72057594054639616	791673868	0	3	72057594054639616	1

```
-- 7. View data with partition number
SELECT TransactionID, TransactionDate, $Partition.pf_OrderDate(TransactionDate) PartitionNo
FROM dbo.PartitionedTransactions
```

Results | Messages

	TransactionID	TransactionDate	PartitionNo
42843	42843	2003-12-31 00:00:00.000	1
42844	42844	2003-12-31 00:00:00.000	1
42845	42845	2004-01-01 00:00:00.000	2
42846	42846	2004-01-01 00:00:00.000	2
:	:	:	:
113442	113442	2004-09-03 00:00:00.000	2
113443	113443	2004-09-03 00:00:00.000	2
113444	113444	2005-01-01 00:00:00.000	3

Querying Partitions

- You can query partitions by using a special function called ***$PARTITION***.
  [*database_name*.] $PARTITION.*partition_function_name*(expression)
- The ***$PARTITION*** function returns a partition number based on the column values for a particular partition function. The most common ways to use this function are …
  - Determine the partition number to which a particular value would correspond.
  - **EX: SELECT $partition.pf_OrderDate ('2004-7-4') as [PartitionNum];**

ROOM310PC1.A...LQuery1.sql* | Object Explorer Details

```
SELECT $partition.pf_OrderDate ('07/01/2004') as [PartitionNum];
```

Results | Messages

	PartitionNum
1	2

  - Restrict a query to a specific partition.
  - **Ex: SELECT * FROM dbo.PartitionedTransactions WHERE $partition. pf_OrderDate (PartitionNumer) = 3**

ROOM310PC1.A...LQuery1.sql* | Object Explorer Details

```
titionedTransactions WHERE $partition. pf_OrderDate (TransactionDate) = 3
```

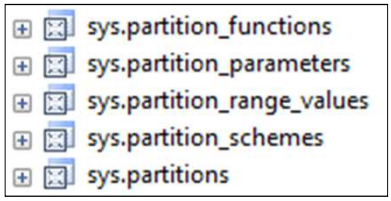
Results | Messages

	TransactionID	ProductID	TransactionDate	TransactionType
1	113444	1	2005-01-01 00:00:00.000	S

```
-- 8. Verify lowest value in each partition
SELECT MIN(TransactionDate) FirstTran, $Partition.pf_OrderDate(TransactionDate) PartitionNo
FROM dbo.PartitionedTransactions
GROUP BY $Partition.pf_OrderDate(TransactionDate)
ORDER BY PartitionNo
```

Results | Messages

	FirstTran	PartitionNo
1	2003-09-01 00:00:00.000	1
2	2004-01-01 00:00:00.000	2
3	2005-01-01 00:00:00.000	3




```
-- 9. Reset database
DROP TABLE dbo.PartitionedTransactions
DROP PARTITION SCHEME ps_OrderDate
DROP PARTITION FUNCTION pf_OrderDate
ALTER DATABASE AdventureWorks REMOVE FILE data1
ALTER DATABASE AdventureWorks REMOVE FILE data2
ALTER DATABASE AdventureWorks REMOVE FILE data3
ALTER DATABASE AdventureWorks REMOVE FILE data4
ALTER DATABASE AdventureWorks REMOVE FILEGROUP fg1
ALTER DATABASE AdventureWorks REMOVE FILEGROUP fg2
ALTER DATABASE AdventureWorks REMOVE FILEGROUP fg3
ALTER DATABASE AdventureWorks REMOVE FILEGROUP fg4
```


What Operations Can Be Performed on Partitioned Data?

- **SWITCH:** Swap a populated table or partition with an empty table or partition by using the SWITCH clause of the ALTER TABLE statement. This technique is commonly used to archive data from a partitioned table or to bulk-insert new data from a staging table.

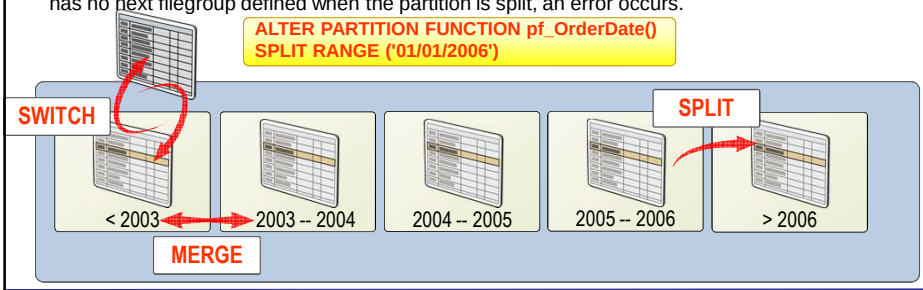
```
ALTER TABLE dbo.PartitionedTransactions
SWITCH PARTITION 1
TO dbo.TransactionArchive
```

- **MERGE:** Combine two adjacent partitions into a single partition. When performing a merge operation, the partition for the boundary value specified in the ALTER PARTITION FUNCTION statement is removed, and the data is merged into the adjacent partition.

```
ALTER PARTITION FUNCTION pf_OrderDate()
MERGE RANGE ('01/01/2003')
```

- **SPLIT:** Insert a boundary in an existing partition to create a new partition. Like a merge, a split operation is performed by using the ALTER PARTITION FUNCTION statement. This creates a new partition and reassigns the data accordingly. The new partition is created on the filegroup designated as the next filegroup in each partition scheme based on the partition function. If a partition scheme has no next filegroup defined when the partition is split, an error occurs.

```
ALTER PARTITION FUNCTION pf_OrderDate()
SPLIT RANGE ('01/01/2006')
```



```
USE MYDB
GO

CREATE PARTITION FUNCTION myPF (int)
AS RANGE LEFT FOR VALUES (50)

ALTER DATABASE MyDB ADD FILEGROUP fg1
ALTER DATABASE MyDB ADD FILEGROUP fg2

ALTER DATABASE MyDB
ADD FILE
( NAME = data1,
  FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\data1.ndf',
  SIZE = 1MB,
  MAXSIZE = 2MB,
  FILEGROWTH = 1MB)
TO FILEGROUP fg1
ALTER DATABASE MyDB
ADD FILE
( NAME = data2,
  FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\data2.ndf',
  SIZE = 1MB,
  MAXSIZE = 2MB,
  FILEGROWTH = 1MB)
TO FILEGROUP fg2

CREATE PARTITION SCHEME myPF
AS PARTITION myPF TO (fg1, fg2);

CREATE TABLE dbo.MyPart (ID int)
ON MyPF(ID)

INSERT INTO dbo.MyPart VALUES (33)
INSERT INTO dbo.MyPart VALUES (44)
INSERT INTO dbo.MyPart VALUES (77)
INSERT INTO dbo.MyPart VALUES (88)
```



```
SELECT * FROM sys.Partitions WHERE [object_id] = OBJECT_ID('dbo.Mypart')
```

Results | Messages

	partition_id	object_id	index_id	partition_number	hobt_id	rows
1	72057594038386688	2105058535	0	1	72057594038386688	2
2	72057594038452224	2105058535	0	2	72057594038452224	2

```
SELECT * FROM MyPart
```

Results

	ID
1	33
2	44
3	77
4	88

```
Update dbo.myPart set ID=99 WHERE ID=33
SELECT * FROM sys.Partitions WHERE [object_id] = OBJECT_ID('dbo.Mypart')
```

Results | Messages

	partition_id	object_id	index_id	partition_number	hobt_id	rows
1	72057594038386688	2105058535	0	1	72057594038386688	1
2	72057594038452224	2105058535	0	2	72057594038452224	3

```
ALTER PARTITION FUNCTION MyPF() MERGE RANGE (50)
SELECT * FROM sys.Partitions WHERE [object_id] = OBJECT_ID('dbo.Mypart')
```

Results | Messages

	partition_id	object_id	index_id	partition_number	hobt_id	rows
1	72057594038452224	2105058535	0	1	72057594038452224	4