SQL Server: Table and Index Partitioning Hands-on Lab

Table of Contents

Content created by

Updated for IEVLT, May 2021

Lab Introduction

Objectives

The goal of these hands-on lab materials is to get an understanding of when to use one of the more advanced features of SQL Server: Table and Index Partitioning.

The intent of these exercises is to provide you with best practices and implementation details for SQL Server 2008 or higher.

As with all software development projects, your production environment may differ from this build and this environment. Be sure to design, implement, and test your final architecture extensively to minimize downtime and data loss.

For the latest details on SQL Server, please visit http://www.microsoft.com/sql/.

After completing these self-paced labs, you will be able to:

- Understand appropriate uses for partitioning
- Create, setup and manage partitioned tables
- Read execution plans to see when a partitioned object is being accessed as well as understand which partitions are being used
- Create indexes on partitioned tables
- Understand when constraints are required within partitioned tables
- Effectively switch data in and out of the partitioned table scenario
- Use the SQL Server Management Studio (SSMS) to manage solutions and projects
- Use SQL Server Management Studio to modify and execute SQLCMD mode scripts
- Use SQL Server Management Studio to execute queries and review their plan of execution

Prerequisites

- Experience with Administration and Optimization tasks in SQL Server
- Experience with SQL Server Management Studio is very helpful
- Familiarity with the Transact-SQL language is also helpful
- Desire to sink your teeth into SQL Server partitioning!

Estimated Time

90 minutes

Setting up for the Lab

This lab was originally created on SQL Server 2008. The database name, expected rows, etc… is based on the SQL Server 2008 version of the AdventureWorks sample database (AdventureWorks2008). The original backup / download used FileStream but this is not needed for this lab exercise. As a result, I've created a new backup / version of the AdventureWorks2008 database that neither requires FileStream nor uses it.

You can download this copy from here (42.2 MB):
http://www.sqlskills.com/resources/AdventureWorks2008Original_NoFS.zip

Create a subdirectory called SQLskills and then unzip the backup there. Once you unzip the SQL Server 2008 R2 backup, the backup can be restored (WITH MOVE) to:

- SQL Server 2008 R2
- SQL Server 2012
- SQL Server 2014
- SQL Server 2016
- SQL Server 2017
- SQL Server 2019

Do NOT restore it now. You'll restore it as part of the lab.

I've mostly tested this lab with the SQL Server 2008 R2 backup restored to SQL Server 2019. However, the entire process is the same across the other versions. If you run into any troubles – do not hesitate to shoot me an email: Kimberly@SQLskills.com. Thanks!

Also, if you're interested in downloading newer versions of the AdventureWorks sample databases, check out the Microsoft Sample Databases page on GitHub – referenced here: https://docs.microsoft.com/en-us/sql/samples/sql-samples-where-are?view=sql-server-ver15. NOTE: There are some differences in the schema as well as the data so a different version of AdventureWorks will affect your results.

Tips on how to successfully complete the lab exercises

IMPORTANT: For best results, read the Tasks column and its associated notes first. Then, proceed to follow the detailed steps for step by step instructions on what to verify, review, execute, etc. Please check off steps as you complete them and execute steps in the order in which they are listed. As a best practice, read the entire step, think about what it is meant to accomplish and then execute it. There are numerous steps that you do NOT immediately execute and instead may have you review and/or alter what you might expect your normal behavior to be in order to show you a feature. There are lots of notes along the way, all of which give you insight into special features, cool tips, and best practices.

Finally, as the complexity of the exercise increases, the need to follow steps in a certain order also increases. Some exercises may fail if steps are missed and worse yet, other exercises might require you to start over if/when steps are missed.

Consider using a pen/pencil to mark off steps as you complete them. Additionally, consider taking a step back from each step to think about what you're trying to accomplish. This will help to minimize errors and increase learning as you may be able to predict certain steps and/or think of additional things to test and learn!

Lab Exercises

Background

The concept of partitioning was not new to SQL Server 2008. In fact, forms of partitioning have been possible in every release. However, without features to aid in creating your partitioning scheme, partitioning it is often extremely cumbersome and underutilized as the design is misunderstood by users and developers unfamiliar with the design. Beginning in SQL Server 7.0, Microsoft has been significantly improving the features related to partitioning and SQL Server 2005's release made the largest advances with an entire feature: table partitioning.

In that release, SQL Server simplified creating table partitions – with both the developer and user – in mind. Most of the manageability benefits relate to managing the sliding window scenario. More specifically, some of the major benefits are:

1. Load data into a new partition of existing partitioned table with minimal disruption in data access in the remaining partitions
2. Load data into a new partition of existing partitioned table with performance equal to loading the same data into a new empty table
3. Delete portion of a partitioned table minimally impacting access to the rest of the table
4. Perform various maintenance operations on per partition basis by rolling partitions in and out of the partitioned table
5. Simplify design of large tables that need to be partitioned for management purposes
6. Improve simplicity in management over all previous releases for partitions

For more information, please read the SQL Server 2005 Partitioned Tables and Indexes whitepaper on MSDN, written by Kimberly L. Tripp. http://msdn.microsoft.com/en-US/library/ms345146(v=SQL.90).aspx.

Objectives

After completing this lab, you will be able to:

- Understand the scenarios for which you would use table partitioning:
- Understand the requirements of table partitioning
    - Partition Functions – Define the logical partition boundaries.

  - Partition Schemes – Map the partitions to physical filegroups within the database. A Partition Scheme always references a Partition Function.

- Understand Range Partitioned Scenarios, specifically managing the Sliding Window Scenario

Scenarios for using partitioning

There are many exciting features related to partitioning, this lab exposes many of them. Expected time to complete the entire lab – while reading documentation, so that you can grasp all of the concepts fully – is 75 minutes. If you continue the lab with reading the whitepaper and executing the 6 scripts associated with the whitepaper – 5 hours.

Primary Use: Data Archiving and Mixed OLTP/Decision Support Scenarios

Biggest Benefit to SQL Server 2008 Partitioning: The ability to "switch out" old data and "switch in" new data – extremely quickly. Range partitions are best when data access is typically decision support data over large periods of time. In this case, you care specifically where the data is located so that only the appropriate partitions are accessed when necessary. Additionally, as transactional data becomes available you will want to add that data in – easily and quickly.

Sliding Window Lab Scenario: To show how easily data can be switched in and out, you will define partitions based on calendar quarters (first quarter is January through March, second quarter is April through June, etc.) using range partitioning. This is most appropriate for decision support scenarios where data access is focused to specific time periods or mixed environments where data pattern usage varies. It is not a requirement of partitioning that the boundaries be equally spaced, nor do they have to follow a specific pattern, such as months or quarters. However, for this lab and for simplicity, this is the pattern that will be used.

Lab Scenario

You are the database administrator for the **AdventureWorks2008** database. You manage an **Orders** table and the performance has suffered greatly as you bring additional OLTP data in as well as when you remove the older unneeded data. Subsequently, all range-based management for decision support has become difficult as the table has become larger. At all times, you want users to have access to one year of order information. However, when each new set of data becomes available, the process of inserting that data and rebuilding the indexes slows overall access to the table for more than 3 hours. You decide to use range-based partitioning to improve the management of data and performance.

In the first part of the exercise, you decide to partition the table for decision support operations based on specific date-based quarters to improve manageability (moving data in and out of the table) and performance. For this lab scenario, the one year's worth of data will cover the four calendar quarters starting with third quarter of 2003 (OrderDate >= Jul 01, 2003) through end of 2nd Quarter 2004 (OrderDate < Jul 01, 2004). In the first part of the exercise, you will define and configure your Partitioned Table scenario and in the second part of the exercise, you will begin

to understand Range Partition Management. In the second half, you will see how partitions are managed when data needs to be removed (the older data – third quarter of 2003) and new data needs to be added (the new data – third quarter of 2004). Regardless of the amount of data moving in and out of your partitioned tables; you can management these changes effectively.

Tasks	Detailed Steps
Open the Partitioning Scripts solution – which contains all 10 scripts needed to work through this lab as well as the whitepaper exercises.	**1.** Start **SQL Server Management Studio**. **2.** Connect to your local instance (where you've restored the AdventureWorks2008 sample database). **3.** Select the **File | Open | Project/Solution** menu item. Navigate to the `Partitioning Lab` directory. Select `PartitioningScripts.ssmssln` and click **Open**. **4.** Once open in **SQL Server Management Studio**, navigate to the **Solution Explorer** window. If this window is not open, select **Solution Explorer** from the **View** drop-down menu. **5.** In the **Solution Explorer** window, notice that there are three projects in one solution. The solution is titled: Solution '`PartitioningScripts`' and the three projects are titled: `Lab Scripts, Whitepaper Scripts and Xtra-ILLExtendedExercises`. For these exercises, you will focus on the project titled: `Lab Scripts`. The "Whitepaper Scripts" project corresponds with the SQL Server 2005 Partitioned Tables and Indexes whitepaper on MSDN; however, this version of the scripts found in this VPC uses a partitioning function that specifies boundaries that fall to the right. The extra "ILL" project is meant for self-paced study and in "instructor-led lab" scenarios (these are often what Kimberly has demo'ed in Partitioning workshops and lectures). Typically, used in lectures related to partitioned views, partitioned tables, and online piecemeal restore. If time permits, consider going through the rest of these scripts at the end of this lab.
Create filegroups and files on which the future partitioned table will reside	**1.** Under the Queries section of the Lab Scripts project, double-click on the file titled: `Script1 - AddFilegroups.sql` to open this into a query window. **Note:** the files will be placed in the `AdventureWorks2008Test` subdirectory on drive C:\. The lab manual references drive C:\ and the actual scripts reference drive D:\SQLskills (as the "root" of all directories). Please change this to YOUR specific drive letter / path. In most production environments, these files would be on separate hard

Tasks	Detailed Steps
	drives – or multiple partitions might be combined within filegroups. For this exercise and for simplicity, all files will be placed in this directory. However, this directory does not exist. In the first part of this script, you will execute the section labeled: **Lab Setup: Step One** to create this directory. To successfully execute this command, you must be in SQLCMD Mode. **2.** Change to allow **SQLCMD Mode**, if not already set. Select **SQLCMD Mode** from the **Query** menu. Notice how some lines appear highlighted in gray. This signifies a SQLCMD command. **3.** Highlight and **Execute** the complete !!mkdir line in **SQLCMD Mode**. `!!mkdir C:\AdventureWorks2008Test` **4.** In the next section of the script (lines 41-49), you will create a test copy of the AdventureWorks2008 backup named AdventureWorks2008Test. Modify the paths as appropriate and then execute this section: `RESTORE DATABASE [AdventureWorks2008Test]` `FROM DISK = N'C:\Ad...Original.BAK'` `WITH  FILE = 1,` `  MOVE N'AdventureWorks2008_Data'` `  TO N'C:\...TestData.mdf'` `  , MOVE N'AdventureWorks2008_Log'` `  TO N'C:\... TestLog.ldf'` `, NOUNLOAD, STATS = 10` `GO` Once the test copy of the database has been restored, you can create the files and filegroups. For this lab, the filegroups are named based on the data they will hold. However, in many "rolling range" scenarios you will want to reuse the same files and filegroups and just cycle through them – always keeping the same files/filegroups but changing what data resides in them. If that's the case, you will want to use generic naming conventions for the filegroups and files. To better see what data resides where and to show management with new files and filegroups (for each quarter), specific date-related names will be used for both the files and the filegroups. **5.** Create four filegroups that will later hold the partitions of your partitioned table. **Execute** the next section of the script to create four filegroups: `ALTER DATABASE AdventureWorks2008Test` `ADD FILEGROUP [2003Q3]` `GO`

Tasks	Detailed Steps

```
ALTER DATABASE AdventureWorks2008Test
ADD FILEGROUP [2003Q4]
GO

ALTER DATABASE AdventureWorks2008Test
ADD FILEGROUP [2004Q1]
GO

ALTER DATABASE AdventureWorks2008Test
ADD FILEGROUP [2004Q2]
GO
```

A filegroup is only a name to a location within a database. The physical location of the data within a filegroup is based on the files that are within a filegroup. A file can only be in one filegroup but a filegroup can have many files. When a filegroup has many files, space is allocated to objects by using a "round-robin" algorithm and essentially these files will be proportionally filled over time – yielding better resource utilization for larger objects – especially when not partitioned. When using a partitioned table strategy, it is more likely that each partition reside in one file as no single partition will warrant multiple files.

6. Create one file in each of the four filegroups. Execute the final section of the script to create these files:

```
ALTER DATABASE AdventureWorks2008Test
ADD FILE
  (NAME = N'RPFile1',
  FILENAME =
N'C:\AdventureWorks2008Test\RPFile1.ndf',
  SIZE = 5MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 5MB)
TO FILEGROUP [2003Q3]
GO

ALTER DATABASE AdventureWorks2008Test
ADD FILE
  (NAME = N'RPFile2',
  FILENAME =
N'C:\AdventureWorks2008Test\RPFile2.ndf',
  SIZE = 5MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 5MB)
TO FILEGROUP [2003Q4]
GO

ALTER DATABASE AdventureWorks2008Test
ADD FILE
  (NAME = N'RPFile3',
```

Tasks	Detailed Steps
	FILENAME = N'C:\AdventureWorks2008Test\RPFile3.ndf', SIZE = 5MB, MAXSIZE = 100MB, FILEGROWTH = 5MB) TO FILEGROUP [2004Q1] GO ALTER DATABASE AdventureWorks2008Test ADD FILE (NAME = N'RPFile4', FILENAME = N'C:\AdventureWorks2008Test\RPFile4.ndf', SIZE = 5MB, MAXSIZE = 100MB, FILEGROWTH = 5MB) TO FILEGROUP [2004Q2] GO **7.** Verify that your AdventureWorks2008Test database has these new filegroups and files and that they're all of the appropriate size. Execute the final batch to review the file and filegroup properties: USE AdventureWorks2008 go sp_helpfile go **8.** Close the file: **Script1 - AddFilegroups.sql.**
Add a new **Orders** table – for testing and comparisons between partitioned and non-partitioned structures	In order to compare plans and differences between partitioned and non-partitioned tables, a copy of a subset of data will be created in a simple non-partitioned table. There is nothing new in this part of the lab – these steps are necessary to create a copy of data and make some interesting modifications for later examples. **1.** Under **Queries** section of **Lab Scripts** project, double-click **Script2 - CreateOrders.sql** **2.** In the **query** window**,** review and then **Execute** this script. You should have 2757 rows in the new **Orders** table. **3.** Close the file: **Script2 - CreateOrders.sql.**

Understanding Partitioned Tables Using a Date Range

In this exercise, you will create a range partition function, a partition scheme, and a partitioned table. Once created, you will load data into the table, add a clustered index and get a feel for how the query plans differ between partitioned and non-partitioned tables.

Understanding the concepts – Range Partition Function

The first step in partitioning a table is to specify the function that you will use to designate how the rows will be directed to the partitions. A range partition always covers the complete range of possible data values – from negative infinity to positive infinity. Because of this, when you specify *n* boundaries, your partitioned object will have *n+ 1* partitions.

In a partition function, you need to define the boundary points. In this scenario, four filegroups have been created. Each filegroup will store one calendar quarter of the **Orders** data. If four boundaries are used, then five partitions will be created (*more on this coming up*).

In the range partitioning syntax, there are two ways to partition data – LEFT or RIGHT. Specifying LEFT or RIGHT determines whether or not the boundary condition is an upper boundary or a lower boundary – in the first or second partition. In other words, if the first value (or boundary condition) of a partition function is 20031001 then the values within that partition will be:

For LEFT

1st partition is all data <= 20031001

2nd partition is all data > 20031001

For RIGHT

1st partition is all data < 20031001

2nd partition is all data => 20031001

If you are using a datetime data type remember that a date with no time implies a 0 time of 12:00am. If LEFT is used with this type of data then you'll end up with Oct 1 12:00am data in the 1st partition and the rest of Oct in the 2nd partition.

Logically, it is best to use beginning values (of the second partition set) with RIGHT and ending values (of the first partition set) with LEFT. The four following clauses create **identical** partitioning structures – for data:

```
RANGE LEFT FOR VALUES ('20030930 23:59:59.997',
                       '20031231 23:59:59.997',
                       '20040331 23:59:59.997',
                       '20040630 23:59:59.997')

RANGE RIGHT FOR VALUES ('20030701 00:00:00.000',
                        '20031001 00:00:00.000',
                        '20040101 00:00:00.000',
                        '20040401 00:00:00.000')
OR
RANGE RIGHT FOR VALUES ('20030701',
                        '20031001',
                        '20040101',
                        '20040401')

As another option, you can also use functions within your partition function.
However, only the actual (the literal) boundary point will be stored.

RANGE LEFT FOR VALUES (dateadd (ms, -3, '20030701'),
                       dateadd (ms, -3, '20031001'),
                       dateadd (ms, -3, '20040101'),
                       dateadd (ms, -3, '20040401'),
```

The datetime data type adds a bit of complexity here but we need to make sure we setup the correct boundary case. Notice the simplicity with RIGHT – this is easier as the default time is 12:00:00.000am. For LEFT there is added complexity due to the datatime datatype. The reason that 23:59:59.997 **MUST** be chosen is that datetime data does not guarantee precision to the millisecond. Instead datetime data is precise to the *nearest* timetick (3.33ms). In the case of 23:59:59.999 this exact timetick is not available and instead the value gets rounded to the nearest timetick which is 00:00:00.000 (12:00:00.000am) of the following day. With this rounding the boundaries will not be defined properly. For datetime data you must use caution with millisecond values. With SQL Server 2008, you can also use new date and time data types; using a data type that stores only a date (and no time) would eliminate this timetick rounding problem. For more details on the new data types, see the books online topic: Date and Time Data Types and Functions (Transact-SQL).

In our example, notice the partition is based on the **OrderDate** column yet the partition function does not specify the name of the column, only the data type. A partition function is more "general" and it not tied directly to a table. The range partition function will be named **OrderDateRangePFN** and will be defined using RIGHT. Because of the simplicity in defining RIGHT-based partition functions with date-related data types, RIGHT is preferred.

```
USE AdventureWorks2008Test
GO
CREATE PARTITION FUNCTION OrderDateRangePFN(datetime)
AS
RANGE RIGHT FOR VALUES ('20030701',
                        '20031001',
                        '20040101',
                        '20040401')
GO
```

There are absolutely no performance or later manageability reasons to choose right over left or visa versa. The most complex part about right and left are in the initial partition function. As you work through the sliding window scenario (i.e. split and merge exercises), consider keeping scratch paper available to draw out your partitioning scheme and where the data resides – and where you want the data to go. This will make visualization (and later automation) a lot easier to understand. In terms of partition function creation, be sure to understand these best practices:

- Always enter a boundary point for each partition you want to create, this will create n+1 partitions
- When creating a RIGHT PARTITION FUNCTION always use the lower boundaries of your data partitions
- When creating a LEFT PARTITION FUNCTION always use the upper boundaries of your data partitions.

Tasks	Detailed Steps
Create the Partition Function – that will later be used by the **Orders** table	**1.** In the **Solution Explorer** window, under **Queries** section of **Lab Scripts** project, double-click: **`Script3 - RangePartitionedTable.sql`**. **2.** In the **query** window, review and then **Execute** the first part of this script – just to create the partition function: `CREATE PARTITION FUNCTION OrderDateRangePFN(datetime) AS RANGE RIGHT FOR VALUES ( '20030701', '20031001', '20040101', '20040401') GO` **NOTE:** Make sure you change database context to **AdventureWorks2008** first!

Understanding the concepts – Partition Scheme

A partition function only defines logical boundary points, not the specific location on which the partitions should reside. A partition scheme maps the partitions to physical locations – specifically filegroups – of your database. If you created a partition function with four boundary points then you will have a portioned object with five partitions. Remember, a partition function must cover the entire range from negative infinity to positive infinity.

Because of the best practices used above, your partition function will not cause record relocation when data is moved in or out of your partitioned object. As a further best practice, one of your partitions will remain empty. If efficient data management in the sliding window scenario is desired, then a LEFT-based partition function will end up with an empty partition on the far right and a RIGHT-based partition function will end up with an empty partition on the far left. This will ensure that no rows move.

There is no need for a special location for the filegroup that will remain empty. For the empty partition, you can use the Primary filegroup.

Tasks	Detailed Steps
Create the Partition Scheme – that will later be used by the **Orders** table	In **Script3 – RangePartitionedTable.sql**, review and execute the second part of this script – just to create the partition scheme, note that comments (shown in green) have been excluded from the code shown here: `CREATE PARTITION SCHEME OrderDatePScheme` `AS` `PARTITION OrderDateRangePFN` `TO ([PRIMARY], [2003Q3], [2003Q4],` `[2004Q1], [2004Q2])` `GO`

Understanding the concepts – Partitioned Table

Once the partitioning function has defined the datasets, and the partitioning scheme has defined which datasets target which filegroups, you can create the partitioned table. To create a partitioned table the table must include a column with the data type used in the creation of the partition function. The ON clause defines which partitioning scheme this table uses (**OrderDatePScheme**) and the column of the table that dictates row to partition (a.k.a. scheme) location.

Because it's likely that your data has constraints that are more restricting than a partition function (remember, the function covers every possible value!), it's a good practice to restrict your data to enforce this property. **Constraints on the base table are not required for partitioned tables to work**; however, they can ensure both data integrity as well as better management if the data is restricted properly. The "current" version of the table will store data from 2003 Q3 to 2004 Q2. To enforce this, a check constraint will be added to the **OrderDate** column.

> **Performance note:** If you are creating a new partitioned table then the table can be created directly on a partition scheme. However, if you are loading existing data from another table/database, it is often better to load data *without* constraints and then add the constraints AFTER the data is loaded AND the indexes are created. Additionally, if your load process also includes data cleansing and/or transformations – you might want to break this process down into multiple, separate steps:
> (1) Load the data into an empty heap structure located on a staging area (this staging area will be reused each month) which is not partitioned
> (2) Perform data consistency checks, constraints, transformations, etc.

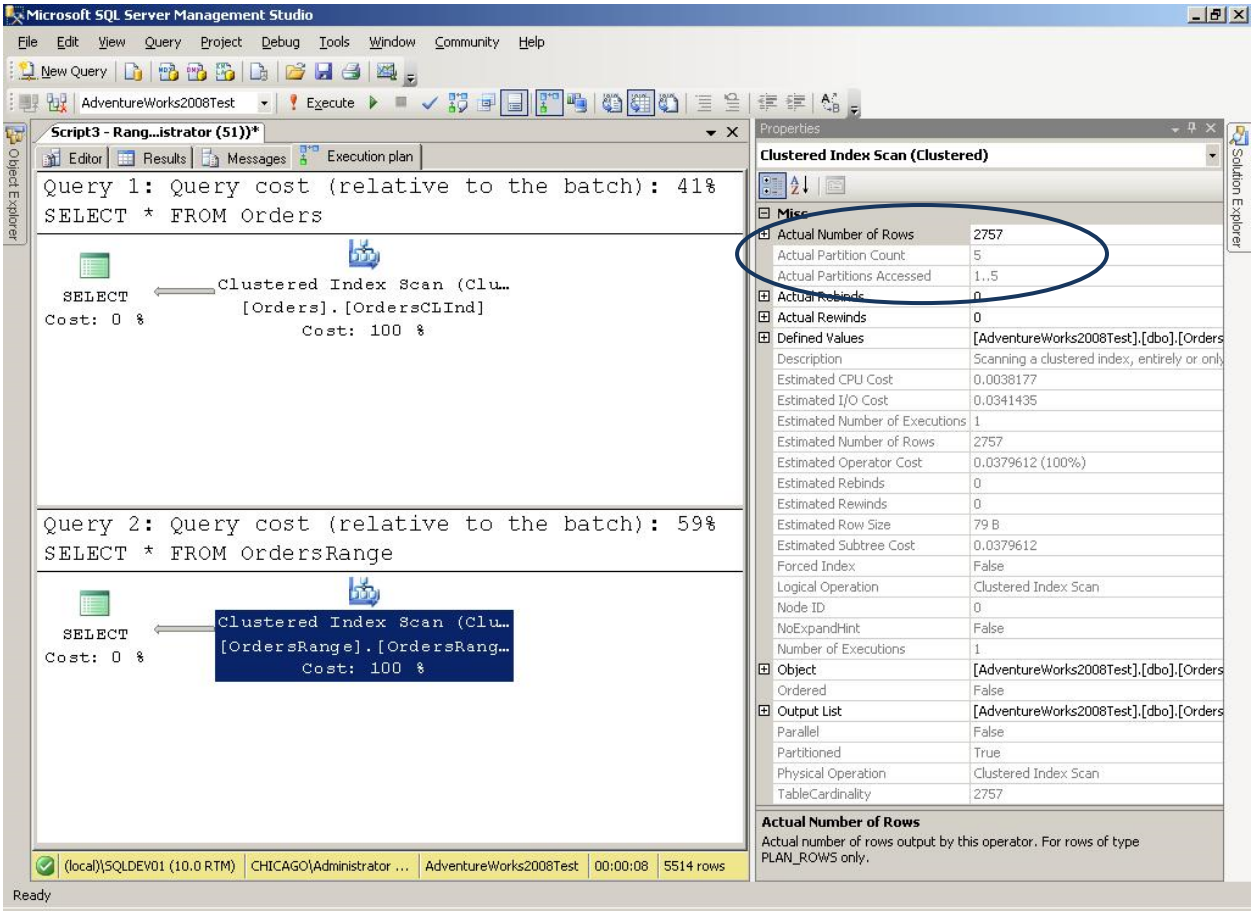
(3) Build the clustered index on the destination filegroup OR if partitioning, the destination partition scheme.

In this example, the tables are small. For simplicity, the constraints and scheme are added to the CREATE TABLE definition and then INSERT…SELECT is used to copy data from another table. For larger tables this would not be ideal. If time permits, consider rewriting the code in Script3 to: create the table first (no constraints, not partitioned), load the data (using INSERT…SELECT) and then add the constraints and "partition" the table. However, this is a bit of a trick question, in order to partition an already existing table, you need to partition by rebuilding (or adding) a clustered index to the table. A heap cannot be partitioned – and kept as a heap. Later in the script you will create a clustered index. If the table had not already been partitioned, then the creation of the clustered index would move the data into a partitioned structure – as a single step.

Tasks	Detailed Steps
Create the Partitioned Table, load the data and verify the row locations using new partition functions	**1.** In `Script3 – RangePartitionedTable.sql`, review and **Execute** the next section of this script – just to create the partitioned object - **OrdersRange**: `CREATE TABLE AdventureWorks2008Test.[dbo].[OrdersRange]` `(` `[OrderID] [int] NOT NULL,` `[EmployeeID] [int] NULL,` `[VendorID] [int] NULL,` `[TaxAmt] [money] NULL,` `[Freight] [money] NULL,` `[SubTotal] [money] NULL,` `[Status] [tinyint] NOT NULL ,` `[RevisionNumber] [tinyint] NULL,` `[ModifiedDate] [datetime] NULL,` `[ShipMethodID] tinyint NULL,` `[ShipDate] [datetime] NOT NULL,` `[OrderDate] [datetime] NOT NULL` `CONSTRAINT OrdersRangeYear` `CHECK ([OrderDate] >= '20030701'` `AND [OrderDate] < '20040701'),` `[TotalDue] [money] NULL` `)` `ON OrderDatePScheme (OrderDate)` `GO` **2.** **Execute** the next section of this script to load data into the **OrdersRange** table by using INSERT…SELECT from the previously created **Orders** table: `INSERT INTO OrdersRange` `SELECT o.[OrderID]` `, o.[EmployeeID]` `, o.[VendorID]` `, o.[TaxAmt]`

Tasks	Detailed Steps
	 `` , o.[Freight] , o.[SubTotal] , o.[Status] , o.[RevisionNumber] , o.[ModifiedDate] , o.[ShipMethodID] , o.[ShipDate] , o.[OrderDate] , o.[TotalDue] FROM dbo.Orders AS o GO ``` To aid in determining where data resides, a new function was created. The breakdown of this function is that you add **$partition.** to the partition function name **function(value)** and then you pass in a value. The value is executed against the partition function and the result is the partition number. **3.** The data has been loaded into the partitions. **Execute** the next section of this script to show each **OrderDate** and the partition number on which the row resides. ``` SELECT OrderDate, $partition.OrderDateRangePFN(OrderDate) AS 'Partition Number' FROM OrdersRange ORDER BY OrderDate GO ``` **4.** In addition to seeing the partition for each individual row, the next query will show the row count in each range as well as the min and max **OrderDate** for each partition. ``` SELECT $partition.OrderDateRangePFN(OrderDate) AS 'Partition Number' , min(OrderDate) AS 'Min Order Date' , max(OrderDate) AS 'Max Order Date' , count(*) AS 'Rows In Partition' FROM OrdersRange GROUP BY $partition.OrderDateRangePFN(OrderDate) ORDER BY 1 GO ```
Create Clustered Indexes on both tables to determine the changes and impact to the execution plan.	In general, a table is typically more optimized when the table is clustered. Many of the best clustered indexes are those that help to improve query performance, minimize fragmentation and offer benefits to certain types of table access. In a range partitioned table scenario (which is date-based), you typically have an ideal clustered index by clustering on your partitioning key. If you cluster by partition key alone then there are some negative side effects. Internally the clustering key must be unique – if you were to create an already unique clustering key then this would eliminate some overhead on INSERT/UPDATE and disk space. An ideal clustering key would be on the **OrderDate** and

Tasks	Detailed Steps
	PurchaseOrderID columns. Since the clustered index will define how the data is stored, it must be created on the partition scheme. **1.** **Execute** the following part of the script to create a clustered index on your partitioned table: `CREATE UNIQUE CLUSTERED INDEX OrdersRangeCLInd` `ON OrdersRange(OrderDate, OrderID)` `ON OrderDatePScheme(OrderDate)` `GO` **2.** For comparisons, **Execute** the following part of the script to create a similar index on the non-partitioned **Orders** table: `CREATE UNIQUE CLUSTERED INDEX OrdersCLInd` `ON Orders(OrderDate, OrderID)` `ON [PRIMARY]` **3.** Execute the following two queries and in the **Execute plan** tab examine the query plan. `SELECT * FROM Orders` `SELECT * FROM OrdersRange` `GO` **4.** In the **Execution plan** tab, what's interesting to note is that the **Properties** window (F4) gives more details than the tooltips ("mousing" over the various icons), so be sure to check both options. Place your cursor over the Clustered Index Scan clause as well as click on it to select it and populate the **Properties** window. In the **Properties** window notice the "Actual Partitions Accessed" of 1..5 and Actual Partition Count of 5.



	If you're interested, there is no difference in the plans generated on this execution versus an execution where the tables do not have clustered indexes (well, except that each of the plans say Table Scan instead of Clustered Index Scan). The only difference is that the clustered index defines the order of the data so instead of seeing "Table Scan" you will now see Clustered Index scan. However, this example does show an important point about the creation of an index over a partitioned table – its simplicity. Once a table has been partitioned, you interact with it just as you would a regular table (in terms of indexes). Instead of having to create indexes for each partition, the partitioned table dictates the logical breakdown (through the partition function) as well as the physical storage location (through the partition scheme). Also note, when creating the clustered index, the **OrderDatePScheme** does **not** need to be specified because the heap structure was already created on the partitioning scheme. If a clustered index does not change the

	definition (i.e. you do not supply an ON clause), then the clustered index follows the same scheme as the table unless. If you were to specify a different partition scheme or a non-partitioned filegroup destination for the clustered index, you would either change the partition scheme or make this table a non-partitioned table. This is useful to know because significant changes (more than just a single partition switch in or out) to a table's structure and location should always be handled through clustered index changes. For example, if you want to make an existing 4 partition table have 8 partitions (and existing data will move) – it's better to create a new partition function and new partition scheme and then rebuild the clustered index on the new scheme than it is to split 4 times (causing record relocation on each split). In the next section, the concepts of the sliding window scenario will be explained further. **5.** Close the file: `Script3 - RangePartitionedTable.sql`.

Understanding the concepts – The Sliding Window Scenario

Each month, when new data becomes available – you need to bring it into your large table. When this is a single table, the process of updating indexes during the load creates poor performance and significant fragmentation. To optimize the load process, you will work with a separate table instead. This "table" – really a soon-to-be-partition in disguise – will be loaded and manipulated independently of the partitioned table. This process allows the existing table to be unaffected and the isolation of the manipulation against the new table to be more efficient. Once ready, you will remove the old data (Q3 of 2003) from the partitioned table (a.k.a. – the data to be archived will be switched out) and then add the new data (Q3 of 2004) to the partitioned table (a.k.a. – the new data will be switched in).

The entire process consists of these general steps:

1. Get the "staging out" table ready (this is for the partition you plan to remove)
2. Get the "staging in" table (and data) ready (this is more complex)
    - Load/transform/cleanse (loading data into a staging heap table in a staging filegroup/area)
    - Build the clustered index on the correct destination filegroup
3. Alter the partition scheme to set next used (based on the filegroup where you created the clustered index in step 1)
4. Split the function to add the new boundary point (this will put the boundary point on the filegroup specified by setting next used)

Note: none of the first 4 steps impact the actual table AND you can prepare them (the staging in/out tables) in any order!

5. Switch "IN" the new data
6. Switch "OUT" the old data

Note: steps 5 and 6 can be in either order – and are FAST (re: metadata only operations)

7. Merge the boundary point (from the emptied [switched out] partition)
8. Backup/remove the old data (drop the table)

In our lab, the range partitioned table has data from Q3 2003 through Q2 2004. For this example, we will process the new quarter's data (Q3 2004) and then simply switch the "staging" table into the partitioned table as a partition. When switching partitions you will find it's a fast process as only metadata is changed – as long as row location and data movement do not occur!

Tasks	Detailed Steps
Build a location for the new data to move into – effectively a staging "in" location – without disrupting the current (and active) **OrdersRange** table.	**1.** In the **Solution Explorer** window, under **Queries** section of **Lab Scripts** project, double-click: **`Script4 - RollingRangeScenario.sql`**. **2.** In the **query** window, create a filegroup into which this data will be loaded and where the data will reside. `ALTER DATABASE AdventureWorks2008Test` `ADD FILEGROUP [2004Q3]` `GO` **Important Concerns:** Depending on filegroup size and the impact you want to your current data set, you have a few options for how you proceed with the switch in and out of data. You could get rid of the old data first and then reuse the existing filegroup for the new data coming it (but then you'd want to have used more generic file and filegroup names) or you can create a new location without reusing the existing file and filegroup. The pro to the latter approach is that we don't need to wait for file creation time. The con to the this approach is that we'd need to remove the quarter before we load the new quarter which means that there would be a window (possible a quite large one) where the "window" of data includes only 3 quarters and not four. Because of features in SQL Server 2008 – including instant initialization – the "wait" time for creating files – even large files – should not be significant. However, your SQL Server must be configured to support instant initialization as it is not enabled by default. **3.** Add a file to the filegroup – following the same pattern you used earlier: `ALTER DATABASE AdventureWorks2008Test` `ADD FILE` `(NAME = N'2004Q3',` `FILENAME =` `N'C:\AdventureWorks2008Test\2004Q3.ndf',` `SIZE = 5MB,` `MAXSIZE = 100MB,` `FILEGROWTH = 5MB)` `TO FILEGROUP [2004Q3]` `GO` **Important Concerns:** Different quarters with more or less data must be sized appropriately and you should always pre-allocate the space so that time is not wasted through autogrowth. Additionally, you may want these files to be on different hard drives. However, for the purposes of this lab –

Tasks	Detailed Steps
	and for simplicity – drive C:\ is going to be used. **4.** Next, you will create a separate non-partitioned table in which to hold the data that will later become the new partition to the partitioned table. This table MUST have the EXACT structure and clustered index of the table that it will become a partition of. Additionally, to ensure a fast "switch" constraints are required in order to restrict this table's data to only the range which will be used within the partition. For optimal performance we will create the table, populate it with data, and then create the clustered index. `CREATE TABLE` `AdventureWorks2008Test.[dbo].[Orders2004Q3]` `(` `[OrderID] [int] NOT NULL,` `[EmployeeID] [int] NULL,` `[VendorID] [int] NULL,` `[TaxAmt] [money] NULL,` `[Freight] [money] NULL,` `[SubTotal] [money] NULL,` `[Status] [tinyint] NOT NULL ,` `[RevisionNumber] [tinyint] NULL,` `[ModifiedDate] [datetime] NULL,` `[ShipMethodID] tinyint NULL,` `[ShipDate] [datetime] NOT NULL,` `[OrderDate] [datetime] NOT NULL` `CONSTRAINT Orders2004Q3MinDate` `CHECK (OrderDate >= '20040701'),` `[TotalDue] [money] NULL` `) ON [2004Q3]` `GO` `ALTER TABLE` `AdventureWorks2008Test.[dbo].[Orders2004Q3]` `ADD CONSTRAINT Orders2004Q3MaxDate` `CHECK (OrderDate < '20041001')` `go` `---------------------------------------------` `-- Populate new table with Q3 2004 data.` `---------------------------------------------` `INSERT INTO` `AdventureWorks2008Test.[dbo].Orders2004Q3` `SELECT o.[PurchaseOrderID]` `, o.[EmployeeID]` `, o.[VendorID]` `, o.[TaxAmt]` `, o.[Freight]` `, o.[SubTotal]` `, o.[Status]` `, o.[RevisionNumber]`

Tasks	Detailed Steps
	`` , o.[ModifiedDate] , o.[ShipMethodID] , o.[ShipDate] , o.[OrderDate] , o.[TotalDue] FROM AdventureWorks2008.Purchasing.PurchaseOrderHeader AS o WHERE o.OrderDate >= '20040701' AND o.OrderDate < '20041001' GO --- -- The table *must* have the same clustered -- index definition! --- CREATE UNIQUE CLUSTERED INDEX Orders2004Q3CLInd ON Orders2004Q3(OrderDate, OrderID) ON [2004Q3] GO ``` **Important Concerns:** Depending on how many rows may or may not meet the constraint requirements (this depends a lot on the validity of the data source), you might want to load the data, clean it up, then add the constraint and then create the clustered index. Also, at this point the new data is ready to be "switched" in. However, if you want to minimize the time between switching the old data out and the new data in (in order to always maintain as close to one year of data as possible), you might want to prepare to "switch" out the old data first.
Build a location for the old data to move to – effectively a staging "out" location.	For switching partitions out you MUST have a table with the EXACT same definition and clustered index created on the same filegroup of the partition you are switching out of the partitioned table. In this case we're going to switch out Q3 2003 – which is on filegroup **2003Q3**. 1. Create the staging table on the appropriate filegroup: ``` CREATE TABLE AdventureWorks2008Test.[dbo].[Orders2003Q3] ([OrderID] [int] NOT NULL, [EmployeeID] [int] NULL, [VendorID] [int] NULL, [TaxAmt] [money] NULL, [Freight] [money] NULL, [SubTotal] [money] NULL, [Status] [tinyint] NOT NULL , [RevisionNumber] [tinyint] NULL, [ModifiedDate] [datetime] NULL, [ShipMethodID] tinyint NULL, [ShipDate] [datetime] NOT NULL, ```

Tasks	Detailed Steps
	[OrderDate] [datetime] NOT NULL, [TotalDue] [money] NULL) ON [2003Q3] GO --- -- The table must have the same clustered -- index definition! --- CREATE UNIQUE CLUSTERED INDEX Orders2003Q3CLInd ON Orders2003Q3(OrderDate, OrderID) ON [2003Q3] GO **Note:** No constraint is necessary for switching data out; the table's data will be coming from the partition's data – which is already restricted. The ONLY constraint that is required in the partitioned table scenario are those on the table that will be switched in. They must be able to know – for certain (trusted constraint) that the data matches the portioning boundary. **2.** In order to switch out a partition you must state the partitioned table's name and the partition number which is being "removed." The partition is only being removed from the partitioned table – the data is NOT deleted. However, the data will NOT be seen via the **OrdersRange** table any longer. The only access to this data is via the **Orders2003Q3** table- into which the partition's data was "switched." ALTER TABLE OrdersRange SWITCH PARTITION 2 TO Orders2003Q3 GO **NOTE:** It might seem challenging to require knowledge of the partition by number but in the rolling range partition scenario the partition being removed will always be 2. Additionally, with a variety of catalog view queries – you can also determine the location of the partition programmatically. There are examples of this in the SQL Server 2005 Partitioned Tables whitepaper. Now you could remove the table with a drop table and completely remove possibly thousands of rows without logging them individually. This provides a very fast mechanism for moving data in and out of ranges. Additionally, a second benefit of having this data isolated in it's own filegroup, is that you effectively get table level restore into a new location if desired. Using partial database restores you can restore just the primary and a subset of filegroups while stilling access the data. Then you can move data into another database. However, you cannot restore this filegroup directly into another database.
Verify that the data within the	**1.** To confirm that we deleted all of the data from partition 2, you can use this special function to see which rows are in which partitions. The partition function exists in the **$partition** namespace and you may

<table>
<tr><th>Tasks</th><th>Detailed Steps</th></tr>
<tr><td>partitioned table no longer includes that which was switched out.</td><td>include it in your query in the format $partition.<function>(<table>.<column>):

<pre>SELECT $partition.OrderDateRangePFN(OrderDate)
              AS 'Parition Number'
   , min(OrderDate) AS 'Min Order Date'
   , max(OrderDate) AS 'Max Order Date'
   , count(*) AS 'Rows In Partition'
FROM OrdersRange
GROUP BY $partition.OrderDateRangePFN(OrderDate)
ORDER BY 1
GO</pre>

2. Verify Data exists in partitions 3, 4 and 5 ONLY</td></tr>
<tr><td>Remove the boundary which is no longer represented by this table and then re-verify the data.</td><td>The previous query should show that you have data ONLY in partitions 3, 4 and 5. So how does the left most partition become 2 again – so that later switches are always switching partition 2 out?

You need to remove the boundary point.

1. Remove the third quarter of 2003 from the OrdersRange partitioned table:

<pre>ALTER PARTITION FUNCTION OrderDateRangePFN()
MERGE RANGE ('20030701')
GO</pre>

2. Now your partitioned table will have only 3 active – and 1 empty – partition. To verify, use the same query above. Notice the partition numbers are now 2, 3, and 4.

<pre>SELECT $partition.OrderDateRangePFN(OrderDate)
              AS 'Parition Number'
   , min(OrderDate) AS 'Min Order Date'
   , max(OrderDate) AS 'Max Order Date'
   , count(*) AS 'Rows In Partition'
FROM OrdersRange
GROUP BY $partition.OrderDateRangePFN(OrderDate)
ORDER BY 1
GO</pre>

3. Data was in partitions 2, 3, 4 but the merge operation removed partition 1. SQL Server therefore renumbers the existing partitions but it does NOT move any data. Only partition numbers have changed. At this point, the filegroups used by this database are unchanged. To see all of the filegroups for this database use the following query:

<pre>SELECT * FROM sys.filegroups</pre></td></tr>
</table>

Tasks	Detailed Steps
	4. However, the filegroup **2003Q3** previously associated with partition 2, is no longer associated with this partitioned table. To see ALL of the filegroups associated with the **OrdersRange** table (even those with no data), use the following query: `SELECT ps.name AS PSName,` `  dds.destination_id AS PartitionNumber,` `  dds.data_space_id AS FileGroup,` `  fg.name AS FileGroupName` `FROM (((sys.tables AS t` `  INNER JOIN sys.indexes AS i` `  ON (t.object_id = i.object_id))` `  INNER JOIN sys.partition_schemes AS ps` `  ON (i.data_space_id = ps.data_space_id))` `  INNER JOIN sys.destination_data_spaces AS dds` `  ON (ps.data_space_id =` `  dds.partition_scheme_id))` `  INNER JOIN sys.filegroups AS fg` `  ON dds.data_space_id = fg.data_space_id` `WHERE (t.name = 'OrdersRange')` `  AND (i.index_id IN (0,1))`
Add the new filegroup to the partition scheme.	In all of the above, you are reviewing the current state of the partitioned table. In order to add a new partition (by splitting one of the existing partitions), you will have to have a place for this new partition to reside. This next step ensures that the next split will use a very specific location for the partition. Again, in order to switch data in or out effectively, you need to have the staging table ready – on the correct filegroup. In this case, the next filegroup to use is **2004Q3** – which is where our data already resides. This step just adds this filegroup to the partition scheme so that it can be used. `ALTER PARTITION SCHEME OrderDatePScheme` `NEXT USED [2004Q3]` `GO`
Alter the table constraints to support the new range of data and switch in our new partition.	Unlike Partitioned Views which rely heavily on constraints to determine what data resides in each table, Partitioned Tables do not require constraints (with the exception of the table that's being switched in). However, for data integrity purposes you might want to constrain your data to a smaller set than what's required by partitioning (which is negative infinity to positive infinity). To do this you need to add constraints on the base table – which are in place. However, the existing constraints restrict the data to only allow **2003Q3** through **2004Q2**. Before we can switch in the new data, we need to allow the correct range. 1. Change the base table constraints to allow 3rd quarter of 2004 as well as not allow 3rd quarter of 2003: `ALTER TABLE OrdersRange` `ADD CONSTRAINT OrdersRangeMax` `  CHECK ([OrderDate] < '20041001')`

Tasks	Detailed Steps

```
go

ALTER TABLE OrdersRange
ADD CONSTRAINT OrdersRangeMin
   CHECK ([OrderDate] >= '20031001')
go

ALTER TABLE OrdersRange
DROP CONSTRAINT OrdersRangeYear
go
```

2. Next, we can allow a new partition to be added to the table:

```
ALTER PARTITION FUNCTION OrderDateRangePFN()
SPLIT RANGE ('20040701')
GO
```

3. Now, you can switch in the new partition:

```
ALTER TABLE Orders2004Q3
SWITCH TO OrdersRange PARTITION 5
GO
```

4. As a final step, verify the data:

```
SELECT $partition.OrderDateRangePFN(OrderDate)
                AS 'Parition Number'
   , min(OrderDate) AS 'Min Order Date'
   , max(OrderDate) AS 'Max Order Date'
   , count(*) AS 'Rows In Partition'
FROM OrdersRange
GROUP BY $partition.OrderDateRangePFN(OrderDate)
ORDER BY 1
GO
```

5. Close the file: **Script4 - RollingRangeScenario.sql.**

Finally, if further partitioning exercises are desired consider walking through the scripts in the Whitepaper Scripts project, also located within this solution. These scripts are updated versions of those provided with the whitepaper and in each script's descriptive header – seen when you open the file – you can see information regarding the changes that were made. There are two scenarios covered by these scripts:

Scenario 1

This covers a more complex Range Partitioned Scenario that includes multiple tables as well as details about joins between two partitioned tables that are aligned. The scripts in your solution use a RIGHT-based partitioning function whereas the whitepaper described the scenario using a LEFT-based partitioning function. The end result is the same but RIGHT-based partitioning functions make more logical sense for date-related date. For this scenario, you need to use the following scripts in this order:

1) RangeCaseStudyScript1-Filegroups.sql

2) RangeCaseStudyScript2-PartitionedTable.sql
3) RangeCaseStudyScript3-JoiningAlignedTables.sql
4) RangeCaseStudyScript4-SlidingWindow.sql

Scenario 2

This covers a Range Partitioned Scenario that simulates List-based Partitioning using Regions. For this scenario, you need to use the following scripts in this order:

1) RegionalRangeCaseStudyScript1-Filegroups.sql
2) RegionalRangeCaseStudyScript2-PartitionedTable.sql

Additional Partitioning Resources

Be sure to review the course materials, links, blog posts, whitepapers, videos, and online courses for more information!

Thanks again!
Kimberly