SQL Server 2005
Partitioned Tables and Indexes

Author: Kimberly L. Tripp, Founder, SQLskills.com

Summary: Although partitioning tables and indexes has always been a design tactic chosen to improve performar larger databases, Microsoft SQL Server 2005 has new features that simplify the design. This whitepaper describes from manually partitioning data by creating your own tables to the preliminary features, which enabled partitionin Server 7.0 and SQL Server 2000, to the true partitioned table features available in SQL Server 2005. In SQL Serve based partitioning features significantly simplify design and administration of partitioned tables while continuing tc The paper's primary focus is to detail and document partitioning within SQL Server 2005 – giving you an understa how to use partitioned tables for the greatest benefit in your VLDB (Very Large Database). Although primarily a VL all databases start out large. SQL Server 2005 provides flexibility and performance while significantly simplifying t maintenance of partitioned tables. Review this document to get detailed information about why you should conside they can offer and finally how to design, implement, and maintain partitioned tables.

Scripts from this Whitepaper:

The scripts and examples used in the code samples for this whitepaper can be found in the SQLServer2005Partitio

Table of Contents

Why do you need Partitioning?
- The History of Partitioning
  - Partitioning Objects manually in releases before SQL Server 7.0
  - Partitioned Views in SQL Server 7.0
  - Partitioned Views in SQL Server 2000
  - Partitioned Tables in SQL Server 2005
- Definitions and Terminology
  - Range Partitions
  - Defining the Partitioning Key
  - Index Partitioning
  - Special Conditions for Partitions – Split, Merge and Switch
- Steps for Creating Partitioned Tables
  - Determine IF Object should be partitioned
  - Determine Partitioning Key and Number of Partitions
  - Determine IF Multiple Filegroups should be used
  - Create filegroups
  - CREATE PARTITION FUNCTION for a Range Partition
  - CREATE PARTITION SCHEME
  - Create the partitioned table
  - Create Indexes: Partitioned or Not?
- Putting it all Together: Case Studies
  - Range Partitioning - Sales Data
  - Joining Partitioned Tables
  - Sliding Window Scenario
  - List Partitioning – Regional Data
- Summary

Why do you need Partitioning?

Before one can talk about how to implement partitioning and the features of partitioning one must first understanc partitions and why might someone consider using them? When you create tables, you design those tables to store entity – i.e. customers or sales. Each table should have attributes that describe only that entity and for customers premise is that all of your customers and all of your sales go into their respective tables. While a single table for e design and understand, it may not be the best for performance, scalability and manageability especially as the tab Partitioning can provide benefits for both large tables (and/or their indexes) and tables which have varying access specifically, through partitioning practices large tables have better scalability and manageability and the use of tab data is simplified when adding or deleting large "chunks" (or ranges) of data.

So what constitutes a large table? The idea of VLDB (Very Large Database) is that the total size of the database is gigabytes or even terabytes but the term does not necessarily specify individual table sizes. A large table is one th desired or one where the maintenance costs have gone beyond pre-defined maintenance periods. Furthermore, a large if one user's activities significantly affect another or if maintenance operations affect other user's abilities. In availability. Even though the server is available, how can you consider your database available when the sales tabl severely degraded or even inaccessible during maintenance for 2 hours per day, per week, or even per month? In downtime is acceptable yet it is often possible to avoid or minimize downtime through better design.

A table whose access patterns vary may also be considered large when sets (or ranges) of rows have very differer Although usages patterns may not always vary (and this is not a requirement for partitioning), when usage patter additional gains. Again, thinking in terms of sales, the current month's data is read-write while the previous montl larger part of the table) is read-only. Large tables where the data usage varies or large tables where the maintena overwhelming can limit the table's ability to respond to varied user requests, in turn limiting both availability and s especially when large sets of data are being used in different ways maintenance operations can end up routinely m Performing maintenance operations on data which does not truly need it – is costly. The costs can be seen in perfc blocking problems, backups (space, time and operational costs) as well as negatively impacting the overall scalabi

Furthermore, if a large table exists on a system with multiple CPUs, partitioning the table can lead to better perfor operations. Large-scale operations across extremely large data sets – typically many million rows – can benefit by operations against individual subsets in parallel. A simple example of performance gains over partitions can be see with aggregations. For example, instead of aggregating a single large table, SQL Server can work on partitions ind aggregate the aggregates. In SQL Server 2005, joins can benefit directly from partitioning; SQL Server 2000 supp operations on subsets yet needed to create the subsets on the fly. In SQL Server 2005, related tables (i.e. Order a partitioned to the same partitioning key and the same partitioning function are said to be aligned. When the optim partitioned and aligned tables are joined, SQL Server 2005 can choose to join the data which resides on the same combine the results. This allows SQL Server 2005 to more effectively use multiple-CPU machines.

So how can partitioning help? Where tables and indexes become very large, partitioning can help by splitting large smaller more manageable chunks (i.e. partitions). The type of partitioning described in this paper is termed horizc horizontal partitioning, large chunks of rows will be stored in multiple separate partitions. The definition of the par defined, and managed – by your needs. Partitioning in SQL Server 2005 allows you to partition your tables based patterns using defined ranges. Finally, SQL Server 2005 offers numerous options for the long-term management o indexes by adding complementary features designed around the new table and index structure.

The History of Partitioning

The concept of partitioning is not new to SQL Server. In fact, forms of partitioning have been possible in every rele features to aid in creating and maintaining your partitioning scheme, partitioning has often been cumbersome and the design is misunderstood by users and developers and the benefits are diminished. However, because of the sig gains inherent in the concept, SQL Server 7.0 began improving the features enabling forms of partitioning through SQL Server 2005 now offers the largest advances through partitioned tables.

Partitioning Objects manually in releases before SQL Ser

In SQL Server 6.5 and earlier, partitioning had to be part of your design as well as built into all of your data access practices. By creating multiple tables and then managing access to the correct tables through stored procedures, you could often improve performance for some operations but at the cost of complexity of design. Each user and c aware of and properly reference the correct tables. Each partition was created and managed separately and views access; however, this solution yielded few performance gains. When a UNIONed view existed to simplify user/appl processor had to access every underlying table in order to determine if data was needed for the result-set. If only underlying tables was needed, then each user and developer needed to know the design in order to reference only

In Figure 1, examine the YearlySales view. Instead of having all sales within one single, large table, you could defi
(SalesJanuary2003, SalesFebruary2003, etc…) and then views for each quarter as well as a View for the entire ye

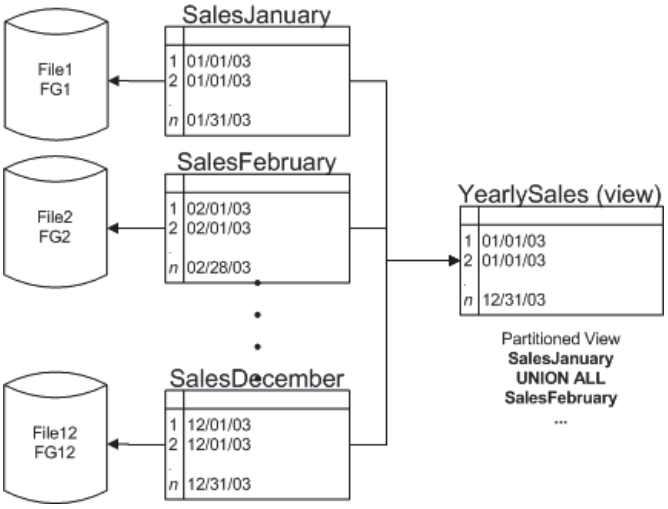


Figure 1: Partitioned Views in SQL Server 7.0/2000

Users who accessed the YearlySales view with the following query will be directed ONLY to the SalesJanuary2003 t

```
SELECT ys.*
FROM dbo.YearlySales AS ys
WHERE ys.SalesDate = '20030113'
```

As long as the constraints are "trusted" and the queries against the view use a WHERE clause to restrict the result
key (the column on which the constraint was defined) then SQL Server will access only the necessary base table. '
constraints where SQL Server is able to guarantee that all data adheres to the properties defined by the constraint
created, the default behavior is to create the constraint WITH CHECK. This setting causes a schema lock to be tak
data can be verified against the constraint. Once the verification validates the existing data, the constraint is adde
is removed further inserts, updates, and deletes must adhere to the constraint in effect. Using this procedure to cr
constraints, developers can significantly reduce the complexity of their design using views without having to know
table in which they were interested. With trusted constraints, SQL Server improves performance by removing the
the execution plan.

Note: A constraint can become "untrusted" in various ways; for instance by performing a bulk-insert and not speci
CHECK_CONSTRAINTS argument. Once a constraint has become untrusted, the query processor will revert to scar
has no way of verifying that the requested data is in fact located in the correct base table.

Partitioned Views in SQL Server 2000

Although SQL Server 7.0 significantly simplified design and improved performance for SELECT statements, it did n
data modification statements; INSERT, UPDATE and DELETE statements were supported only against the base tabl
the views which UNIONed the tables. SQL Server 2000 allows data modification statements to also benefit from th
capabilities of SQL Server 7.0. By allowing data modification statements to use the same partitioned view structur
modifications to the appropriate base table through the view. In order to configure this properly there are addition
partitioning key and its creation; however, the basic principals are the same in that not only will SELECT queries b
appropriate base tables but the modifications will be as well. For details on the restrictions, setup, configuration ar
partitioning in SQL Server 2000, please refer to the Partitioning Whitepaper at:
http://msdn.microsoft.com/library/techart/PartitionsInDW.htm.

Partitioned Tables in SQL Server 2005

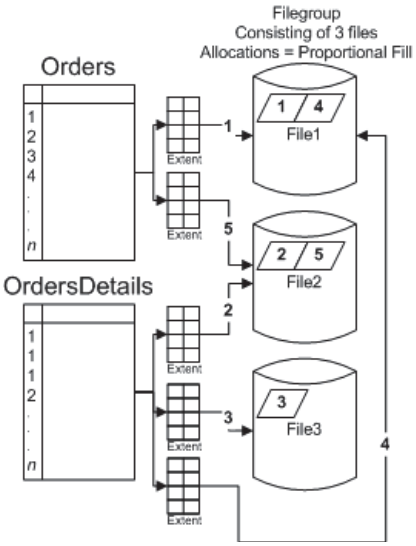
While the improvements in SQL Server 7.0 and SQL Server 2000 significantly enhanced performance using partitic
simplify the administration, design or development of a partitioned data set. Using partitioned views, all of the bas
view is defined) must be created and managed individually. Application design is easier and users benefit by not n
base table to directly access but administration is complex as there are numerous tables to manage and data inte

For example, take a current month of OLTP (Online Transaction Processing) data that needs to be moved into an a end. The latter table (to be used for read-only queries) is a single table with one clustered index and two non-clus load of 1GB (into the already indexed and active single table) creates blocking with current users as the table and, fragmented and/or locked. Additionally, the loading process will take a significant amount of time as the table and maintained as each row comes in. There are ways to speed up the bulk load; however, these can directly affect all sacrificing concurrency for speed. If this data were isolated into a newly-created (empty) table the load could occu are possible) and then the indexes could be created after load (and index creation can be parallelized). Often you times or better by using this scheme. In fact, by loading into an unindexed [heap] table you can take advantage o multiple data files in parallel or loading multiple chunks from the same file (defined by starting and ending row pos partitioning gives you this more granular control and does not restrict you to having all of the data in one location indexes – and no real ability to control any aspect of it at a more granular level. A functional partitioning strategy previous releases by dynamically creating and dropping tables and modifying the UNION view. However; in SQL S more elegant; you can simply "switch in" the newly-filled partition(s) as an extra partition of the existing partition any old partition(s). From end to end, the complete process takes only a short period of time and can be further ir bulk loading and parallel index creation. More importantly, the partition is manipulated outside of the scope of the on the actual table until the partition is added. The result is that adding a partition typically occurs within only a fe

Even better is the case when data needs to be removed. If one database only needs to see a "sliding-window" set data is ready to be migrated in (for example the current month) then the oldest data (maybe the parallel month fr can be removed. In this same example, you will likely see several orders of magnitude better performance improv partitioning. While this may seem extreme, consider the difference; when all of the data is in a single table the de oldest data), requires row-by-row manipulation of the table as well as its associated indexes. The process of deleti significant amount of log activity and does not allow log truncation for the length of the delete (remember the dele commit transaction; however, you can control the size of the transaction by performing multiple deletes – where p requires a [potentially much] larger log. To remove the same amount of data – by removing the specific partition f all that must be done is a removal of the partition (which is a meta data operation) and then drop or truncate the

Moreover, without knowing how to best design partitions one might not be aware that the use of filegroups in conj ideal for implementing partitioning. Filegroups allow you to place individual tables on different physical disks. If a s files using filegroups then the actual physical location of data could not be predicted. For systems where parallelisr Server improves performance by using all disks more evenly through filegroups and specific placement is not as cr

In Figure 2, there are three files in a single filegroup. Two tables: Orders and OrderDetails have been placed or tables are placed on a filegroup SQL Server proportionally fills the files within the filegroup by taking extent (6 eight 8K Pages) allocations from each of the files as space is needed for the objects within the filegroups. Whe OrderDetails tables are created, the filegroup is empty. When an order comes in data is input into the Orders t and one row per line item is input into the OrderDetails table. SQL Server allocates an extent to the Orders tab another extent to the OrderDetails table from File2. It is likely that the OrderDetails table will grow more quick and the next few allocations will go to the next table needing space. As OrderDetails grows, it will get the next SQL Server continues to "round robin" through the files within the filegroup. In the diagram below, follow each from each extent to the appropriate filegroup – the extents are allocated as space is needed and each is numb



With partitioned tables in SQL Server 2005, a table can be designed (using a "function" and a "scheme") such that same partitioning key are placed directly on (and will always go to) a specific location. The function defines the pa which partition the first value should be placed. In the case of a LEFT partition function, the first value will act as a first partition. In the case of a RIGHT partition function, the first value will act as a lower boundary in the second p many more details regarding partition functions later within this paper. Once the function is defined, a partition scl define the physical mapping of the partitions to their location within the database – based on a partition function. the same function (but not necessarily the same scheme) rows that have the same partitioning key will be groupe called alignment. By aligning rows with the same partition key from multiple tables to exist in filegroups on the sa disks, SQL Server can, if the optimizer chooses, work with only the necessary groups of data (from each of the tat alignment two partitioned tables or indexes must have some correspondence between their respective partitions. "equivalent" partition functions and some relation with respect to the partitioning columns. Two partition functions when:

- Both partition functions use the same number of arguments and partitions
- The partitioning key used in each function is of equal type (includes length, precision and scale if applicable, a
- The boundary values are equivalent (including the LEFT/RIGHT boundary criteria)

  NOTE: Even when two partition functions are designed to align data, you may end up with an unaligned index the same column as the partitioned table.

Collocation is a stronger form of alignment where two aligned objects are joined with an equi-join predicate where partitioning column. This becomes important in the context of a query, subquery or another similar construct wher may occur. Collocation is valuable because queries that join tables on the partition columns are in general much fa Orders and OrderDetails tables described above. Instead of filling the files proportionally, you can create a partitio three filegroups. When defining the Orders and OrderDetails tables you will define them to use that same scheme. having the same value for the partition key) will be placed within the same file thereby isolating the necessary dat related rows from multiple tables are partitioned in the same manner, SQL Server can join the partitions without h an entire table or multiple partitions (if the table were using a different partitioning function) for matching rows. Ir are not only aligned (because they use the same key) but they are said to be storage aligned because the same d same files.

The following diagram shows that two objects can use the same partition scheme and all data rows with the same up on the same filegroup. When related data is aligned, SQL Server 2005 can effectively work on large sets in par for January (for both the Orders and OrderDetails tables) is on the first filegroup and February data on the second

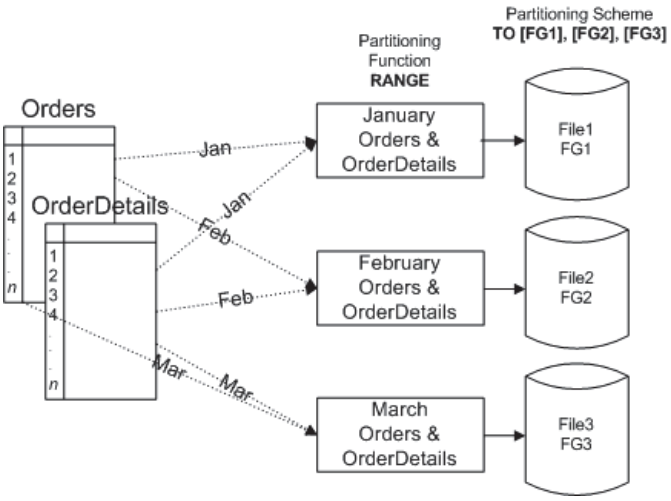


Figure 3: Storage Aligned Tables

SQL Server allows partitioning based on ranges. Tables as well as indexes can use the same scheme for better alig significantly improves overall performance, but what if the usage of the data changes over time? What if an additi Administrative simplicity in adding partitions, removing partitions and managing partitions outside of the partitione design goals for SQL Server 2005.

SQL Server 2005 has simplified partitioning with administration, development and usage in mind. Some of the per manageability benefits are:

- Simplify design and implementation of large tables that need to be partitioned for performance or manageabil
- Load data into a new partition of existing partitioned table with minimal disruption in data access in the remai
- Load data into a new partition of existing partitioned table with performance equal to loading the same data ir

Definitions and Terminology

To implement partitions in SQL Server 2005 you must be familiar with a few new concepts, terms and syntax. In p was always both a physical and a logical concept, yet with SQL Server 2005 Partitioned Tables and Indexes you ha how and where you store a table. In SQL Server 2005, tables and indexes can be created with the same syntax as single tabular structure that is placed on the DEFAULT filegroup or a user-defined filegroup. Additionally, in SQL Se indexes can also be created on a partitioning scheme. The partitioning scheme maps the object to a filegroup or p To determine which data goes to the appropriate physical location(s), the partitioning scheme uses a partitioning f function defines the algorithm to be used to direct the rows and the scheme associates the partitions with their ap (i.e. a filegroup). In other words, the table is still a logical concept but its physical placement on disk can be radica releases; the table can have a scheme.

Range Partitions

Range partitions are table partitions defined by specific and customizable ranges of data. Range partition's bounda developer – and can be changed if data usage patterns change. Typically, these ranges are date based or based or of data.

The primary use of range partitions is for data archiving, decision support (when often only specific ranges of data example, only a given month or quarter) and for combined OLTP and DSS (Decision Support Systems) where data cycle of a row. The biggest benefit to SQL Server 2005 Partitioned Tables and Indexes is the ability to manipulate data, especially related to archiving and maintenance. With range partitions, old data can be archived and new da extremely quickly. Range partitions are best suited when data access is typically for decision support over large ra you care where the data is specifically located so that only the appropriate partitions are accessed when necessary transactional data becomes available you will want to add that data in – easily and quickly. Range partitions are in define as you will need to define the boundary conditions for each of the partitions. Additionally, you will create a s partition to a filegroup(s). However, they often have a consistent pattern to them so once defined they will likely b programmatically (see Figure 4).

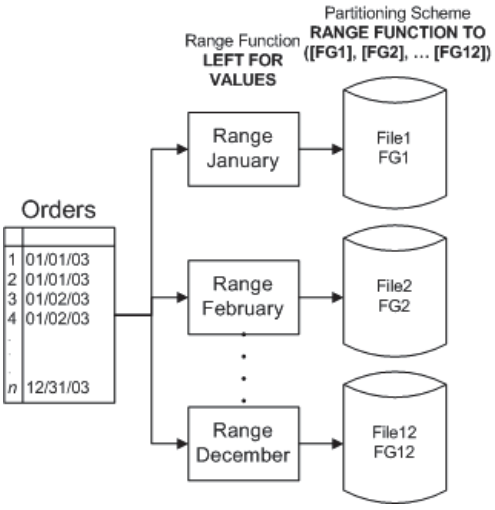


Figure 4: Range Partitioned Table - 12 Partitions.

Defining the Partitioning Key

The first step in partitioning tables and indexes is to define the data on which the partition is "keyed." The Partitio column in the table and must meet certain criteria. The partition function defines the data type on which the logica based. The physical placement of data is determined by the partition scheme. In other words, the scheme maps th that maps the data to specific file(s) and therefore disks. The scheme always uses a function to do this – if the fur partitions then the scheme must use five filegroups. The filegroups do not need to be different however; you will g when you have multiple disks and preferably multiple CPUs. When the scheme is used with a table, you will define as an argument for the partition function.

For Range Partitions the data set is divided by a logical and data-driven barrier. In fact, the data partitions may no However, the usage of the data dictates a range partition because the pattern in which this table is used defines sp

Index Partitioning

In addition to partitioning a table's data set, you can partition indexes. Indexes can be partitioned using the same
or not. However, it is often an ideal combination for performance to partition both the table and its indexes using t
both the indexes and the table use the same partitioning function and the same partitioning columns (in the same
index are said to be aligned. If an index is created on an already partitioned table, SQL Server automatically align
table's partitioning scheme unless the index is explicitly partitioned differently. When a table and its indexes are al
can move partitions in and out of partitioned tables more effectively as all of the related data and indexes are divi
algorithm.

When the tables and indexes are defined with not only the same partition function but also the same partition sch
be storage aligned. In general, having the tables and indexes on the same file or filegroup can often benefit where
parallelize an operation across partitions. In the case of storage alignment and multiple CPUs, SQL Server can hav
directly on a specific file or filegroup – and know that there are no conflicts in data access because all data require
index lookup will be colocated on the same disk. This allows more processes to run in parallel without interruption

For more details, refer to the BOL topic: Special Guidelines for Partitioned Indexes.

Special Conditions for Partitions – Split, Merge and Switc

To aid in the usage of partitioned tables are several new concepts related to partition management. Because partit
tables that need to scale to support better throughput, it is possible that the number of partitions chosen when th
created will need to change over time. Using the ALTER TABLE statement with the new "split" option, you can add
table. When a partition is "split," data may be moved to the new partition; however, for best performance rows sh
scenario illustrating how this works will be described later in this paper.

Conversely, if you want to remove a partition you would perform a switch and merge. In the case of range partitio
made by stating which boundary point should be removed. Where only a specific period of data is needed and dat
a regular basis (for example, monthly) you might want to archive one partition of data (the earliest month) after a
partition in. For example, you might choose to have one year of data available and each month-end you will switch
then switch out the earliest month thereby differentiating between the current month's read/write OLTP data and t
only analysis type of data. However, before you merge a boundary point you should switch out its associated data
could be an expensive operation. There is a specific flow to this process to make it the most efficient. The concept
is somewhat complex at first glance, however a simple scenario will show you the concepts with a logical flow.

In this scenario, you are keeping a year's worth of read-only data available. Currently the table holds data from Se
last month – August 2004. The current month of September 2004 is in another database, optimized for OLTP perfo
version of the table, there are 13 partitions. Twelve partitions which contain data (September 2003 through Augus
partition which is empty (remember that a range partition includes the extreme left as well as the extreme right).
data integrity by using a check constraint to require that the OrderDate be on or after September 1, 2003 and bef
the constraint will effectively keep the last partition empty.

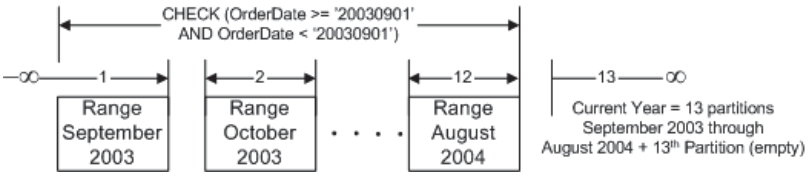


Figure 5: Range Partition Boundaries – Before Data Load/Archive

When October begins (in the OLTP database), we are ready to move September's data to the partitioned table use
process of switching the tables in and out is a very fast process and the preparation work can be performed outsid
This scenario is explained in depth in the case study coming up but the idea is that you will use "staging tables" th
partitions within the partitioned table (switching a table in to become a partition) or hold the older data of a table
to become a standalone table).

In this process, you will switch out (Figure 6) a partition of a table – to a non-partitioned table **within the same f**
non-partitioned table already exists within the same filegroup (and this is critical for success), SQL Server can ma
metadata change. Switching out a partition can occur within seconds as SQL Server solely has to make the metad
location as opposed to running a delete that might take hours and create blocking in large tables. Once this partiti
will still have 13 partitions; the first (oldest) partition is now empty and the last (most recent, also empty) partitio

To remove the first (oldest) partition (September 2003) you will use the new "merge" option (Figure 7) with ALTEF boundary point, you effectively remove the boundary point and therefore remove a partition. This will also reduce into which data loads to n-1 (in this case 12). Merging a partition should be a very fast operation when no rows ha case, because the first partition is empty no rows need to move (from the first to the second partition). If the first and the boundary point is "merged" then rows will have to move from the first to the second partition and this cou operation. However, in the common "sliding window" scenario this should never be the case as you will always me into an active partition and no rows will move.

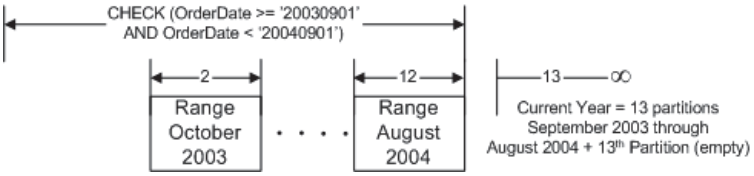


Figure 7: Merging a Partition

Finally, the new table has to be switched in to the partitioned table. Remember, in order for this to be performed a metadata change, all of the work to load and build indexes has to have already occurred on a new table – outside partitioned table. To switch in the partition, you will first split the last (most recent, empty) range into two partitio need to update the table's constraint to allow the new range. Once again, the partitioned table will have 13 partiti the sliding window scenario with a LEFT partition function will always remain empty.

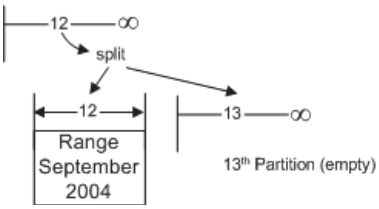


Figure 8: Splitting a Partition

Finally, the newly loaded data is ready to be switched in to the 12th partition – the September 2004 data.

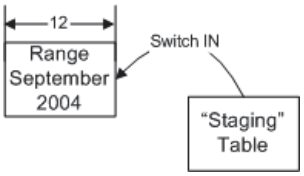


Figure 9: Switching a partition IN

The result of the table is:

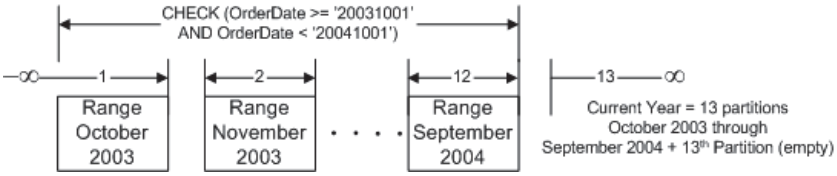


Figure 10: Range Partition Boundaries – After Data Load/Archive

While these are only concepts it is important to know that split and merge are handled through ALTER PARTITION handled through ALTER TABLE. Additionally, only one partition can be added or removed at a time. If a range parti have more than one partition added or removed then a more optimal way of rebalancing the data (rather than reb for each split) can be performed by creating a new partitioned table – using a new partition function and partition "moving" the data over to the new partitioned table. To "move" the data, first copy the data using INSERT *newtab oldtable* and then drop the original tables. Be careful not to lose data by preventing user (or other) modifications v running.

For more details, refer to the Books Online topics: ALTER PARTITION FUNCTION and ALTER TABLE.

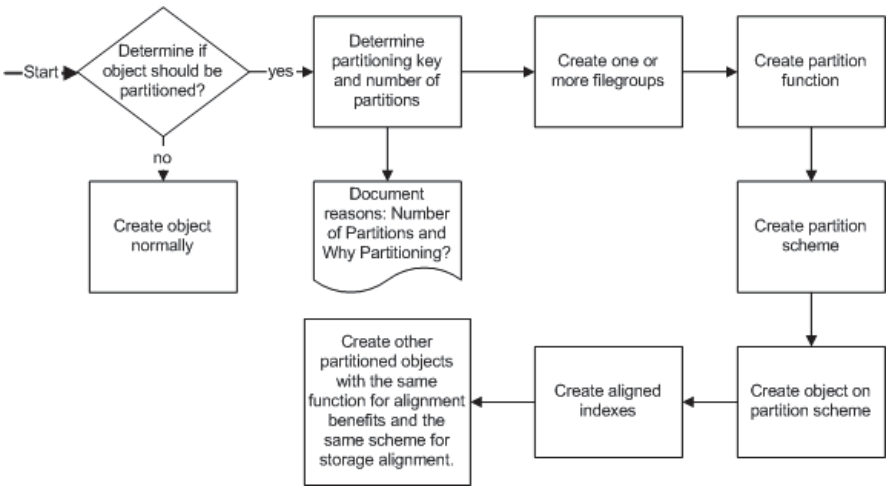


Figure 11: Steps to Create a Partitioned Table or Index

Determine IF Object should be partitioned

As described above this is the first and most important step. Not every table should be partitioned. Although partit the application, partitioning adds administrative overhead and complexity to the implementation of your objects. W great benefits, you may not want to consider it when your table is small. So what constitutes small v. large? Your and maintenance requirements as well as whether or not your tables perform acceptably already. These are all fac whether an existing table should be partitioned. For example, archiving a range of data out of a single sales table of time – minutes (or more) to add new rows in (via INSERT…SELECT) and minutes (or more) to remove old data these create an unnecessary burden on system performance and resources then you should consider partitioning, an administrative burden without significant gain.

Determine Partitioning Key and Number of Partitions

If you are trying to improve performance and manageability for large subsets of data and there are defined access partitioning mechanism. If your data has logical groupings – and you often work with only a few of these defined s define your ranges so the queries are isolated to work with only the appropriate data (i.e. partition).

For more details, refer to the BOL topic: Designing Partitioned Tables and Indexes.

Determine IF Multiple Filegroups should be used

For both performance purposes and easier maintenance, you should use filegroups to separate your data. The nun partially dictated by hardware resources; generally you want to have the same number of filegroups that you have these filegroups will reside on different disks. However, this primarily only pertains to systems where analysis tenc the entire data set. When you have multiple CPUs, SQL Server can process multiple partitions in parallel and there the overall processing time of large complex reports and analysis. In this case, you can have the benefit of paralle switching partitions in and out of the partitioned table.

Create filegroups

If you want to place a partitioned table on multiple files for better IO balancing, you will need to create a filegroup of one or more files and each partition must map to a filegroup. A single filegroup can be used for multiple partitio management – such as for more granular backup control – you should design your partitioned tables wisely so tha grouped data resides on the same filegroup. Using ALTER DATABASE, you add a logical filegroup name – to which create a filegroup named 2003Q3 for the AdventureWorks database, use ALTER DATABASE:

```
ALTER DATABASE AdventureWorks ADD FILEGROUP [2003Q3]
```

Once a filegroup exists then you use ALTER DATABASE to add files to the filegroup.

```
ALTER DATABASE AdventureWorks
```

```
  FILEGROWTH = 5MB)
TO FILEGROUP [2003Q3]
```

A table can be created on a file(s) specifying a filegroup in the ON clause of CREATE TABLE. However, a table cann filegroups without partitioning that table. To create a table on a single filegroup you use the ON clause of CREATE partitioned table you must first have a logical mechanism for the partitioning. Even though you may think of partit logical partitions with a single table in mind, SQL Server allows you to reuse the partition definition for other objec concept of a partition from a table you create and name a partition structure through a partition function. Therefo partitioning a table is to create the partition function.

CREATE PARTITION FUNCTION for a Range Partition

Range partitions must be defined with boundary conditions. Moreover, all boundaries must be included; a range pa include all values even though the table may be *(and should be)* more restricted through a CHECK constraint. No of the range – can be eliminated. Additionally, since data is likely to be switched in and out of the partition you wil empty partition – in which you can repeatedly split – in order to make room for a new partition of data to "switch" partition will always remain empty, as it will always be waiting to include future rows – which you will switch in pe

In a range partition, you will need to define the boundary points first. If you are defining five partitions, only four needed. To partition the data into five groups you will define the four boundary values for the partitions and then is in either in the first (LEFT) or the second (RIGHT) partition. You will always have one partition (at the extreme r function or at the extreme left for a RIGHT partition function) that will not have a defined boundary point explicitly confusing but a partition function includes all data values (-∞ on the far left and ∞ to the far right). By specifying L whether or not the boundary condition is an upper boundary in the first partition (LEFT) or a lower boundary in se other words if the first value (or boundary condition) of a partition function is '20001001' then the values within th be:

For LEFT

1st partition is all data <= '20001001'

2nd partition is all data > '20001001'

For RIGHT

1st partition is all data < '20001001'

2nd partition is all data => '20001001'

Since range partitions are likely to be defined on datetime data, you must be aware of the implications. Using date BOTH a date and a time. A date with no defined value for time implies a "0" time of 12:00am. If LEFT is used with will end up with Oct 1 12:00am data in the 1st partition and the rest of Oct in the 2nd partition. Logically, it is bes (of the second partition set) with RIGHT and ending values (of the first partition set) with LEFT. The three followin identical partitioning structures:

```
RANGE LEFT FOR VALUES ('20000930 23:59:59.997',
                            '20001231 23:59:59.997',
                            '20010331 23:59:59.997',
                            '20010630 23:59:59.997')
OR
RANGE RIGHT FOR VALUES ('20001001 00:00:00.000',
                                  '20010101 00:00:00.000',
                                  '20010401 00:00:00.000',
                                  '20010701 00:00:00.000')
OR
RANGE RIGHT FOR VALUES ('20001001', '20010101', '20010401', '20010701')
```

Note Using the datetime data type does add a bit of complexity here but you need to make sure you setup th Notice the simplicity with RIGHT – this is easier as the default time is 12:00:00.000am. For LEFT the added co precision of the datetime datatype. The reason that 23:59:59.997 MUST be chosen is that datetime data does the millisecond. Instead, datetime data is precise within 3.33 milliseconds. In the case of 23:59:59.999, this e available and instead the value is rounded to the nearest time tick that is 12:00:00.000am of the following da

To store one fourth of the Orders data in the four active partitions (each will represent one calendar quarter) and c
later use (again, as a placeholder for rolling data in and out of the partitioned table), use a LEFT partition function
conditions:

```
CREATE PARTITION FUNCTION OrderDateRangePFN(datetime)
AS
RANGE LEFT FOR VALUES ('20000930 23:59:59.997',
                       '20001231 23:59:59.997',
                       '20010331 23:59:59.997',
                       '20010630 23:59:59.997')
```

Remember, four defined boundary points creates 5 partitions – with one empty partition to the extreme right wher
defined as LEFT and one empty partition to the extreme LEFT when the partition function is defined as RIGHT. Rev
by this partition function by reviewing the sets as follows:

Boundary point '20000930 23:59:59.997' as LEFT (sets the pattern):

The LEFT most partition will include all values <= '20000930 23:59:59.997'

Boundary point '20001231 23:59:59.997':

The SECOND partition will include all values > '20000930 23:59:59.997' but <= '20001231 23:59:5

Boundary point '20010331 23:59:59.997':

The THIRD partition will include all values > '20001231 23:59:59.997' but <= '20010331 23:59:59.

Boundary point '20010630 23:59:59.997':

The FORTH partition will include all values > '20010331 23:59:59.997' but <= '20010630 23:59:59

Finally, a FIFTH partition will include all values > '20010630 23:59:59.997'.

Regardless of LEFT or RIGHT, a range partition function must include all values to infinity in both directions. With a
the last value will define the last absolute partition value but there must be a place for all values – meaning that a
for the data rows greater than this value. There will always be one additional partition at the end for LEFT range p
additional partition at the beginning for RIGHT range partitions.

CREATE PARTITION SCHEME

Once you have created a partition function you must associate it with a partition scheme to direct the partitions to
you define a partition scheme, you must make sure to name a filegroup for EVERY partition, even if multiple partit
same filegroup. For the range partition created previously (OrderDateRangePFN) there are 5 partitions, the last – a
be created in the PRIMARY filegroup. There is no need for a special location for this partition because it will never c

```
CREATE PARTITION SCHEME OrderDatePScheme
AS
PARTITION OrderDateRangePFN
TO ([2000Q3], [2000Q4], [2001Q1], [2001Q2], [PRIMARY])
```

Note: If all partitions will reside in the same filegroup then a simpler syntax can be used as follows:

```
CREATE PARTITION SCHEME OrderDatePScheme
AS
PARTITION OrderDateRangePFN
ALL TO ([PRIMARY])
```

Create the partitioned table

Now that both the logical (the partition function) and physical (the partition scheme) structures are defined the ta
advantage of them. The table defines which "scheme" should be used and the scheme defines the function. To tie
you must specify the column on which the partitioning function should apply. Range partitions always map to exac
table and it should match the datatype of the boundary conditions defined within the partition function. Additionall
specifically restrict the data set to a more limited set (rather than from –infinity to positive infinity) then a check c
as well.

```
      [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] <= '20040630 11:59:59.997'),
      [TotalDue] [money] NULL
)
ON OrderDatePScheme (OrderDate)
GO
```

Create Indexes: Partitioned or Not?

By default, indexes created on a partitioned table will also use the same partitioning scheme and partitioning colu
index is aligned with the table. It is not required, but often desired to have the table and its indexes aligned. Align
allows for easier management and administration especially when working with the sliding window scenario. Howe
to be aligned. Indexes can be created using different partitioning functions or not partitioned at all.

For example, when creating unique indexes, the partitioning column must be one of the key columns; this will ens
appropriate partition must be verified to guarantee uniqueness. Therefore if you need to partition a table on one c
create unique index on a different column then they cannot be aligned; the index may be partitioned on the uniqu
column unique key then this may be any of the key columns) or not partitioned at all. Be aware that this index wi
created when switching data in and out of the partitioned table.

Note If you plan to load a table with existing data and add indexes to it immediately you can often get better pe
a non-partitioned, un-indexed table and then create the indexes to partition the data after the load. By defining a
partition scheme, you will effectively partition a table after the fact. This is also a great way of partitioning an exis
same table as a non-partitioned table and create the clustered index as a partitioned clustered index, replace the
table with a single filegroup destination and create the clustered index on the partition scheme after the data is lo

Putting it all Together: Case Studies

In having looked at concepts, benefits and small snippets of code related to partitioning – you have most of the pr
However, for each step, there are specific settings and options that are available and in certain cases, a variety of
section will help you put everything together.

Range Partitioning - Sales Data

Sales data often varies in usage. The current month's data is transactional data; prior month's data is heavily use
Analysis is often done for monthly, quarterly and/or yearly ranges of data. Because different analysts may want to
different data at the same time – partitioning will better isolate their activity. In this scenario, the active data com
locations and is delivered in two standard format ASCII files. All files are placed on a centrally located fileserver n
first day of each month. Each file ranges in size but averages roughly 86,000 sales (Orders) per month. Each orde
therefore the OrderDetails files averages 226,180 rows. Each month roughly 25 million new Orders and 64 million
added and the historical analysis server maintains two year's worth of data active for analysis. Two year's worth o
million Orders and just over 1.5 billion OrderDetails rows. Because analysis is often done comparing months withi
same month within the prior year – range partitioning is chosen. The boundary for each range is monthly.

Using the steps described in Figure 11: Steps to Create a Partitioned Table, we have decided to partition the table
based on OrderDate. Looking at the requirements for this new server, analysts tend to aggregate and analyze up t
of data or up to 3 months of the current and past year (for example, January through March 2003 with January th

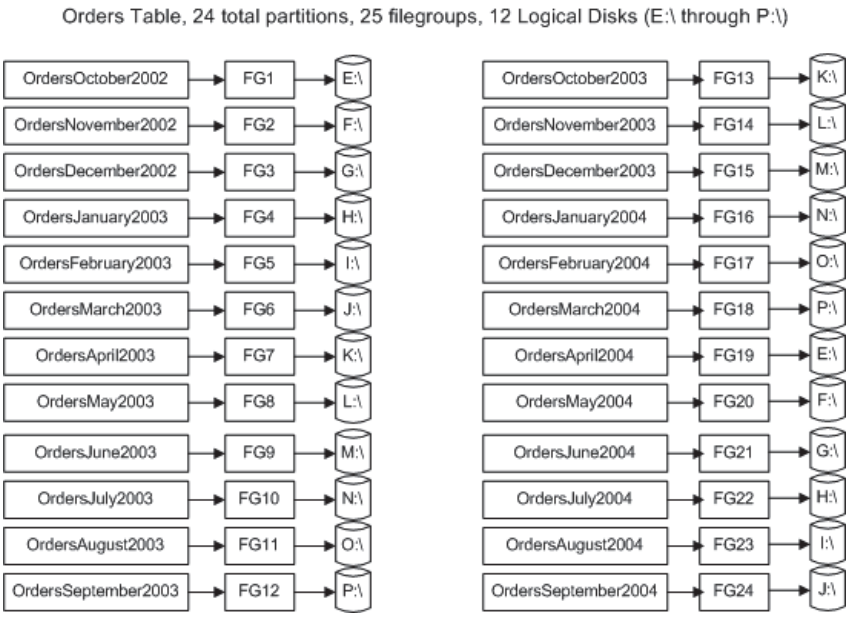


Figure 12: Partitioned Table Orders

Each of the 12 logical drives is in a RAID 1+0 configuration therefore the total number of disks needed for Orders 48. However, the SAN supports 78 disks and the other 30 are used for the transaction log, TempDB, system datab smaller tables such as Customers (9 million) and Products (386,750 rows), etc. Both the Orders and the OrderDet same boundary conditions and the same placement on disk; in fact, they will use the same partitioning scheme. T two logical drives [Drive E:\ and F:\] in Figure 13) is that the data for Orders and OrderDetails will reside on the s months:

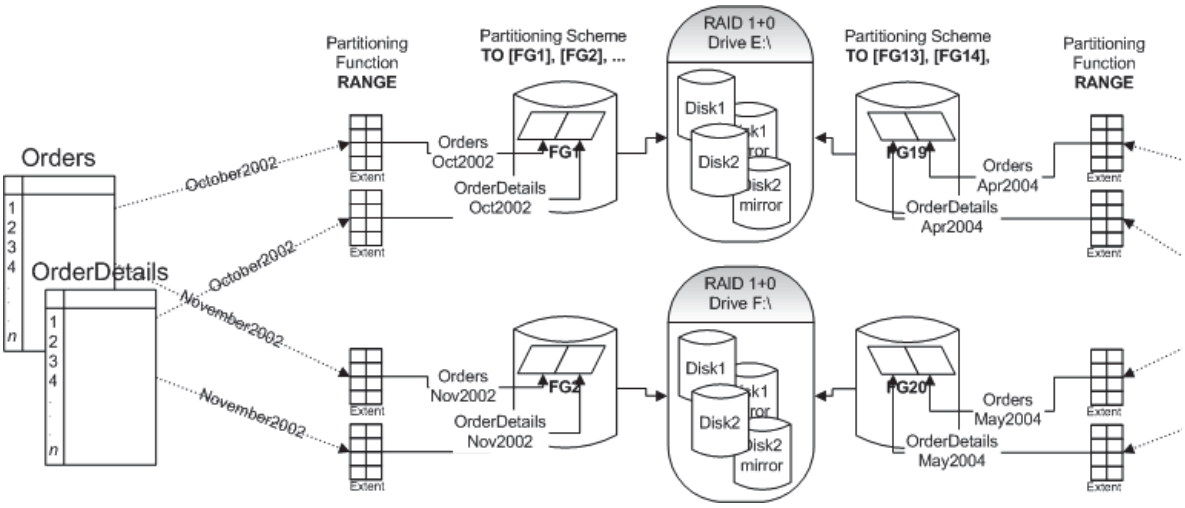


Figure 13: Range Partition to Extent Placement on Disk Arrays

While seemingly complex, this is quite simple to create. The hardest part in the design of the partitioned table is t the large number of sources – 283 stores must have a standard mechanism for delivery. On the central server, ho Orders table and one OrderDetails table to define. In order to create both tables as partitioned tables we must firs function and the partition scheme. A partition scheme defines the physical placement of partition to disk so the file well. In this table, filegroups are necessary, so the next step is to create the filegroups. Each filegroup's syntax is but all twenty-four filegroups must be created. See the RangeCaseStudyFilegroups.sql script for a complete script filegroups (note: you cannot run this script without the appropriate drive letters; however, the script includes a "se modified for simplified testing. You can change the drive letters/locations to a single drive to test and learn the syn adjust the file sizes to MB instead of GB and consider a smaller number for initial size, depending on available disk and filegroups will be created for the SalesDB. Each will have the same syntax – with the exception of location, file name:

```
  FILEGROWTH = 5GB)
TO FILEGROUP [FG1]
GO
```

Once all twenty-four files and filegroups have been created, you are ready to define the partition function and the
verify your files and filegroups use sp_helpfile and sp_helpfilegroup, respectively.

The partition function will be defined on the OrderDate column. The data type used is datetime and both tables wi
OrderDate in order to partition both tables on this value. In effect, the partitioning key value (if both expect to be
key value) will be duplicated information; however, it is necessary for the alignment benefits and (in most cases) s
narrow column (the datetime date type is 8 bytes). As described in *CREATE PARTITION FUNCTION for a Range Pa*
a range partition function where the first boundary condition will be the in the LEFT (first) partition.

```
CREATE PARTITION FUNCTION TwoYearDateRangePFN(datetime)
AS
RANGE LEFT FOR VALUES ('20021031 23:59:59.997',       -- Oct 2002
                       '20021130 23:59:59.997',      -- Nov 2002
                       '20021231 23:59:59.997',      -- Dec 2002
                       '20030131 23:59:59.997',      -- Jan 2003
                       '20030228 23:59:59.997',      -- Feb 2003
                       '20030331 23:59:59.997',      -- Mar 2003
                       '20030430 23:59:59.997',      -- Apr 2003
                       '20030531 23:59:59.997',      -- May 2003
                       '20030630 23:59:59.997',      -- Jun 2003
                       '20030731 23:59:59.997',      -- Jul 2003
                       '20030831 23:59:59.997',      -- Aug 2003
                       '20030930 23:59:59.997',      -- Sep 2003
                       '20031031 23:59:59.997',      -- Oct 2003
                       '20031130 23:59:59.997',      -- Nov 2003
                       '20031231 23:59:59.997',      -- Dec 2003
                       '20040131 23:59:59.997',      -- Jan 2004
                       '20040229 23:59:59.997',      -- Feb 2004
                       '20040331 23:59:59.997',      -- Mar 2004
                       '20040430 23:59:59.997',      -- Apr 2004
                       '20040531 23:59:59.997',      -- May 2004
                       '20040630 23:59:59.997',      -- Jun 2004
                       '20040731 23:59:59.997',      -- Jul 2004
                       '20040831 23:59:59.997',      -- Aug 2004
                       '20040930 23:59:59.997')      -- Sep 2004
GO
```

Because both the extreme left and extreme right boundary cases are included, this partition function actually crea
table will maintain a 25th partition that will remain empty. No special filegroup is needed for this empty partition, a
reside in it. To ensure that no data resides within this 25th filegroup a constraint will restrict the table's data. To di
appropriate disks a partition scheme is used to map the partition to the filegroup. The partition scheme will use ar
for each of the 24 filegroups that will contain data and will use the PRIMARY filegroup for the 25th and empty parti

```
CREATE PARTITION SCHEME [TwoYearDateRangePScheme]
AS
PARTITION TwoYearDateRangePFN TO
```

A table can be created with the same syntax as previous releases support – using the default filegroup or a user-d
non-partitioned table) or using a scheme (to create a partitioned table). As for which is best (even if this table will
table) depends on how the table will be populated and how many partitions you are going to manipulate. Populatir
building the clustered index on it is likely to provide better performance than loading into a clustered table. Additic
machine with multiple CPUs you can load data into the table in parallel and then build the indexes in parallel. For t
create the table normally and then load the existing data through INSERT…SELECT statements. To create the Ord
table you specify the partition scheme in the ON clause of the table. The Orders table is created with the following

```
CREATE TABLE SalesDB.[dbo].[Orders]
(
      [PurchaseOrderID] [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] NULL
            CONSTRAINT OrdersRangeYear
CHECK ([OrderDate] >= '20021001'
         AND [OrderDate] < '20041001'),
      [TotalDue] [money] NULL
) ON TwoYearDateRangePScheme(OrderDate)
GO
```

Since the OrderDetails table is also going to use this scheme and must include the OrderDate the OrderDetails tab
following syntax:

```
CREATE TABLE [dbo].[OrderDetails](
      [OrderID] [int] NOT NULL,
      [LineNumber] [smallint] NOT NULL,
      [ProductID] [int] NULL,
      [UnitPrice] [money] NULL,
      [OrderQty] [smallint] NULL,
      [ReceivedQty] [float] NULL,
      [RejectedQty] [float] NULL,
      [OrderDate] [datetime] NOT NULL
            CONSTRAINT OrderDetailsRangeYearCK
                  CHECK ([OrderDate] >= '20021001'
                         AND [OrderDate] < '20041001'),
      [DueDate] [datetime] NULL,
      [ModifiedDate] [datetime] NOT NULL
            CONSTRAINT [OrderDetailsModifiedDateDFLT]
                  DEFAULT (getdate()),
      [LineTotal]  AS (([UnitPrice]*[OrderQty])),
      [StockedQty]  AS (([ReceivedQty]-[RejectedQty]))
```

```
      SELECT o.[PurchaseOrderID]
                  , o.[EmployeeID]
                  , o.[VendorID]
                  , o.[TaxAmt]
                  , o.[Freight]
                  , o.[SubTotal]
                  , o.[Status]
                  , o.[RevisionNumber]
                  , o.[ModifiedDate]
                  , o.[ShipMethodID]
                  , o.[ShipDate]
                  , o.[OrderDate]
                  , o.[TotalDue]
      FROM AdventureWorks.Purchasing.PurchaseOrderHeader AS o
            WHERE ([OrderDate] >= '20021001'
                         AND [OrderDate] < '20041001')
GO

INSERT dbo.[OrderDetails]
      SELECT      od.PurchaseOrderID
                  , od.LineNumber
                  , od.ProductID
                  , od.UnitPrice
                  , od.OrderQty
                  , od.ReceivedQty
                  , od.RejectedQty
                  , o.OrderDate
                  , od.DueDate
                  , od.ModifiedDate
      FROM AdventureWorks.Purchasing.PurchaseOrderDetail AS od
            JOIN AdventureWorks.Purchasing.PurchaseOrderHeader AS o
                        ON o.PurchaseOrderID = od.PurchaseOrderID
            WHERE (o.[OrderDate] >= '20021001'
                         AND o.[OrderDate] < '20041001')
GO
```

Now that the data has been loaded into the partitioned table, you can use a new built-in system function to deterr
which the data will reside. The following query is helpful as it returns the following information about each partitio
many rows exist within each partition, the minimum and maximum OrderDate. A partition that does not contain rc
this query.

```
SELECT $partition.TwoYearDateRangePFN(o.OrderDate)
                  AS [Partition Number]
      , min(o.OrderDate) AS [Min Order Date]
      , max(o.OrderDate) AS [Max Order Date]
      , count(*) AS [Rows In Partition]
FROM dbo.Orders AS o
GROUP BY $partition.TwoYearDateRangePFN(o.OrderDate)
ORDER BY [Partition Number]
```

```
      , max(od.OrderDate) AS [Max Order Date]
      , count(*) AS [Rows In Partition]
FROM dbo.OrderDetails AS od
GROUP BY $partition.TwoYearDateRangePFN(od.OrderDate)
ORDER BY [Partition Number]
GO
```

Finally, once the data has been populated you can build the clustered index and create the foreign key reference b
Orders. In this case, the clustered index will be defined on the Primary Key as you will also identify both of these t
key (and for OrderDetails you will add the LineNumber to the index for uniqueness). When created, the default be
partitioned tables is to align the index with the partitioned table on the same scheme; the scheme does not need t

```
ALTER TABLE Orders
ADD CONSTRAINT OrdersPK
      PRIMARY KEY CLUSTERED (OrderDate, OrderID)
GO

ALTER TABLE dbo.OrderDetails
ADD CONSTRAINT OrderDetailsPK
      PRIMARY KEY CLUSTERED (OrderDate, OrderID, LineNumber)
GO
```

The complete syntax, specifying the partition scheme would be as follows:

```
ALTER TABLE Orders
ADD CONSTRAINT OrdersPK
      PRIMARY KEY CLUSTERED (OrderDate, OrderID)
      ON TwoYearDateRangePScheme(OrderDate)
GO

ALTER TABLE dbo.OrderDetails
ADD CONSTRAINT OrderDetailsPK
      PRIMARY KEY CLUSTERED (OrderDate, OrderID, LineNumber)
      ON TwoYearDateRangePScheme(OrderDate)
GO
```

Joining Partitioned Tables

When aligned tables are joined, SQL Server 2005 has the option to join the tables in one step or in multiple steps
partitions are joined first and then the subsets are added together. Moreover, regardless of how the partitions are
evaluates whether or not some level of partition elimination is possible.

Partition Elimination

In the following query, data is queried from the Order and OrderDetails tables created in the previous scenario. Th
information from the third quarter only. Typically, the third quarter includes slower months for order processing; h
months were some of the largest months for Orders. In this case, we want to see if there are any interesting trenc
quantities ordered and their order dates – but only for the third quarter. To ensure that aligned partitioned tables l
elimination when joined, you must be sure to state each table's partitioning range. In this case, since the Orders t
composite key of both OrderDate and OrderID the join between these tables will show that OrderDate must be equ
The SARG (Search Argument) will be applied to both partitioned tables. The query to retrieve this data is:

```
SELECT o.OrderID, o.OrderDate, o.VendorID, od.ProductID, od.OrderQty
```

Shown in figure 14, there are some key elements for which to look when reviewing either the actual or the estima (using SQL Server Management Studio), when hovering over either of the tables being accessed you will see eithe Executions" or "Number of Executions." In this case, we are looking at one quarter – or three months – worth of d own partition and therefore there are three executes in looking up this data – one for each table.

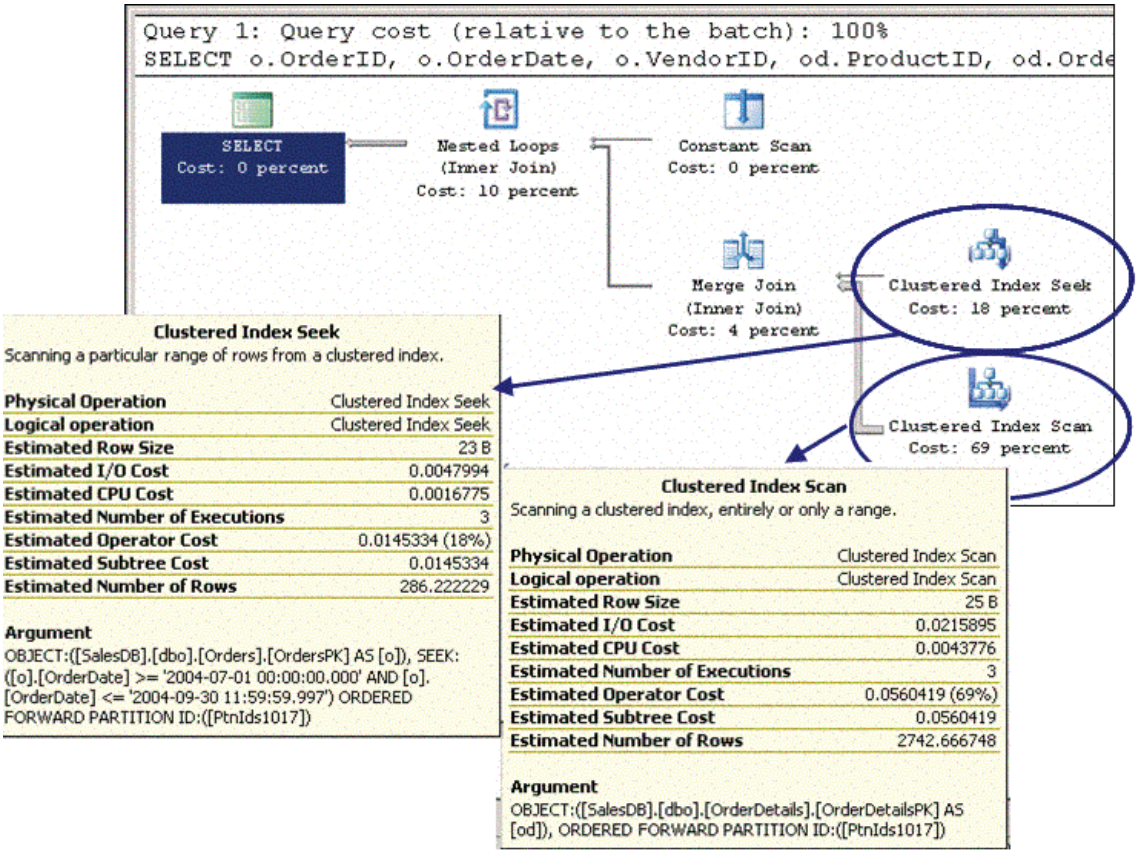


Figure 14: Number of Executes

Shown in figure 15, SQL Server is eliminating all unnecessary partitions and choosing only those that contain the **PARTITION ID:([PtnIds1017])** within the Argument section. You may wonder from where the PtnIds1017 expr hover over the Constant Scan at the top of the showplan you will see that it shows an Argument of **VALUES(((21** represents the partition numbers.

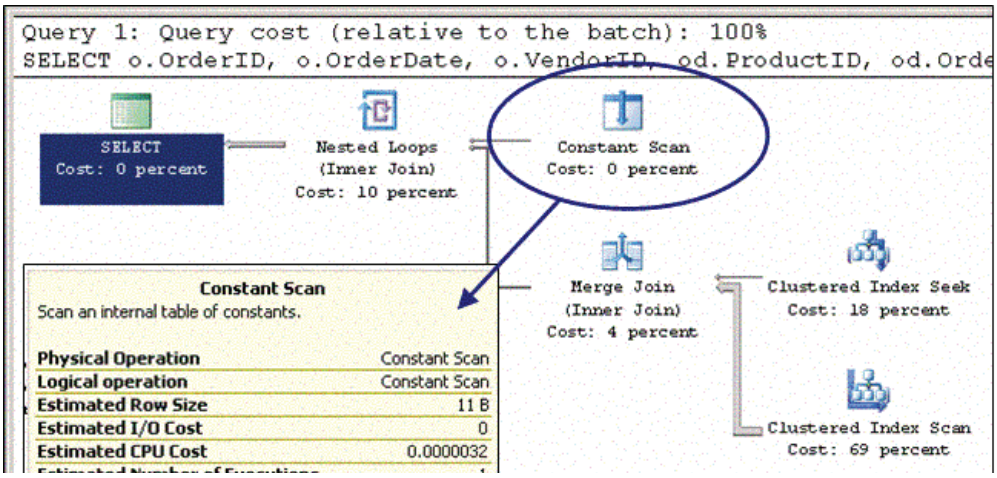


Figure 15: Number of Executes

To verify what data exists on those partitions and only those partitions you can use a slightly modified version of t
to access the new built-in system functions for partitions.

```
SELECT $partition.TwoYearDateRangePFN(o.OrderDate)
                  AS [Partition Number]
      , min(o.OrderDate) AS [Min Order Date]
      , max(o.OrderDate) AS [Max Order Date]
      , count(*) AS [Rows In Partition]
FROM dbo.Orders AS o
WHERE $partition.TwoYearDateRangePFN(o.OrderDate) IN (21, 22, 23)
GROUP BY $partition.TwoYearDateRangePFN(o.OrderDate)
ORDER BY [Partition Number]
GO
```

Moreover, at this point, you can recognize partition elimination graphically. An additional optimization technique ca
tables and indexes – specifically if they are aligned with tables to which you are joining. SQL Server may perform
each of the partitions first.

Pre-Joining Aligned Tables

Within the same query, SQL Server is not only eliminating partitions but also executing the joins between the rem
individually. In addition to seeing the number of executes for each table access you should also notice the informa
"merge" join. If you hover over the merge join, you can see that the merge join was executed three times as well.

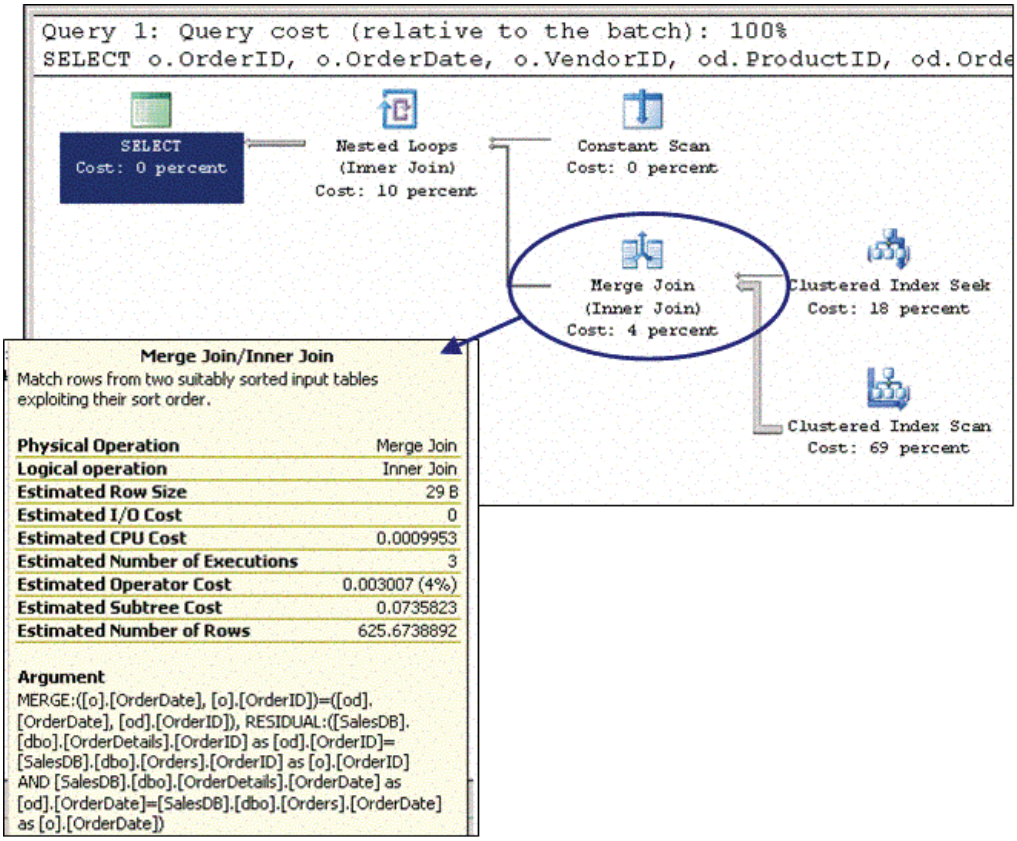


Figure 16: Joining Partitioned Tables

order to switch a partition in or out of a table quickly, the switch must change metadata ONLY. In other words, the destination – i.e., a partition in disguise) must be created on the same filegroup being switched in or out. If you pl same filegroups, FG1 in this case, then you will need to determine how to handle the space and archiving requiren have a minimal amount of time where the table does not have a complete two years worth of data AND you have the current data (October 2004) into FG1 without removing the data to be archived (October 2002). However, if y space to keep both the current and the month to be archived then you will need to switch the old partition out firs

Regardless, archiving should be easy – and is probably already being done. A good practice for archiving is to bacl immediately after the new partition is loaded and switched in (not just before you plan to switch it out). The reaso were a failure on the RAID array, the filegroup could be restored rather than having to rebuild/reload the data. In the database was only recently partitioned – you could have performed a full backup after the partitioning structur database backup is not your only option however, there are numerous backup strategies which can be implemente many offer better granularities for backup and restore). For future backups because so much of the data is **not** ch the individual filegroups after they are loaded. In fact, this should become part of your rolling partition strategy. F to the Books Online topic: "File and Filegroup Backups" in Administering SQL Server.

Now that the strategy is in place, you need to understand the exact process and syntax. The syntax and number c complex however each month the overall process will be identical. All that will change are the dates. By using dyn can easily automate this process. The basic steps are:

- Manage the staging table for the partition that will be switched IN
- Manage the second staging table for the partition that will be switched OUT
- Roll the old data OUT and the new data IN to the Partitioned Table
- Drop the staging tables
- Backup the filegroup

As for the syntax and best practices, each step is detailed in the following sections. Additionally, each step contain automating this process through dynamic SQL execution.

Manage the Staging Table for the partition that will be switched IN

1. Create the staging table – this is the table that is really a future partition in disguise. This staging table must h restrict its data to only valid data for the soon-to-be partition of the table. However, for better performance yo data into an unindexed heap with no constraints and then add the constraint (see step 3) WITH CHECK before the partitioned table.

```
CREATE TABLE SalesDB.[dbo].[OrdersOctober2004]
(
   [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,
   [TotalDue] [money] NULL
) ON [FG1]
GO
```

In automation: this table will be easy to create as it will always be the current or very-recently-completed m your process runs detecting month is easy with built-in functions such as DATENAME(m, getdate()). The table' the existing table – so the primary change for each month is to the table name. However, you could use the sa

```
DECLARE @Month                  nchar(2),
         @Year                  nchar(4),
         @StagingDateRange       nchar(10)
SELECT @Month = N'11', @Year = N'2004'
SELECT @StagingDateRange = @Year + @Month + N'01'
SELECT dateadd(ms, -2, @StagingDateRange)
```

The table will need to be recreated each month, as it will need to reside on the filegroup where the data is swit determine the appropriate filegroup with which to work, use the following system table query combined with th shown earlier. Specify any date within the range being rolled out. This is the partition (and filegroup) in which performed. The underlined sections will need to be changed for your specific table, partitioning function and sp

```
SELECT ps.name AS PSName,
         dds.destination_id AS PartitionNumber,
         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 = 'Orders') AND (i.index_id IN (0,1)) AND
dds.destination_id = $partition.TwoYearDateRangePFN('20021001')
```

2. Load the staging table with data. If the files are consistent then this process should be handled through BULK In automation: this process is the most complex to automate. You will need to make sure that all files have be want to load them in parallel. To control this process you might consider a table that keeps track of which files where the files are located. You can create a SQL Agent job that checks for files every few minutes, picks up th executes multiple bulk insert statements.

3. Once the data is loaded, you can add the constraint. In order for the data to be "trusted", the constraint must The WITH CHECK setting is the default so it does not need to be specified but it is important NOT to say WITH

```
ALTER TABLE SalesDB.[dbo].[OrdersOctober2004]
WITH CHECK
ADD CONSTRAINT OrdersRangeYearCK
   CHECK ([OrderDate] >= '20041001'
         AND [OrderDate] <= '20041031 23:59:59.997')
GO
```

4. Index the staging table – it must have the same clustered index as the table in which it will become a partitior

```
ALTER TABLE [OrdersOctober2004]
ADD CONSTRAINT OrdersOctober2004PK
   PRIMARY KEY CLUSTERED (OrderDate, OrderID)
ON [FG1]
GO
```

```
CREATE TABLE SalesDB.[dbo].[OrdersOctober2002]
(
   [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,
   [TotalDue] [money] NULL
) ON [FG1]
GO
```

2. Index the staging table – it must have the same clustered index as the table in which it will become a partitior become this table).

```
ALTER TABLE [OrdersOctober2002]
ADD CONSTRAINT OrdersOctober2002PK
   PRIMARY KEY CLUSTERED (OrderDate, OrderID)
ON [FG1]
GO
```

Roll the old data OUT and the new data IN to the Partitioned Table

1. Switch the old data out – into the second staging table.

```
ALTER TABLE Orders
SWITCH PARTITION 1
TO OrdersOctober2002
GO
```

2. Alter the partition function to "remove" the boundary point for October 2002.

```
ALTER PARTITION FUNCTION TwoYearDateRangePFN()
MERGE RANGE ('20021031 23:59:59.997')
GO
```

3. This also removes the association between the filegroup and the partition scheme. In other words, FG1 is no l partition scheme. Since you will roll the new data through the same existing 24 partitions then you will need tc used" partition. In other words, this will be the next partition to use.

```
ALTER PARTITION SCHEME TwoYearDateRangePScheme
              NEXT USED [FG1]
GO
```

4. Alter the partition function to "add" the new boundary point for October 2004.

```
ALTER PARTITION FUNCTION TwoYearDateRangePFN()
```

extending the date instead of dropping and re-creating the constraint. For now, only one constraint exists (Ord future dates, there will be two constraints.

```
ALTER TABLE Orders
ADD CONSTRAINT OrdersRangeMaxOctober2004
   CHECK ([OrderDate] < '20041101')
GO

ALTER TABLE Orders
ADD CONSTRAINT OrdersRangeMinNovember2002
   CHECK ([OrderDate] >= '20021101')
GO

ALTER TABLE Orders
DROP CONSTRAINT OrdersRangeYearCK
GO
```

6. Switch the new data in – from the first staging table.

```
ALTER TABLE OrdersOctober2004
SWITCH TO Orders PARTITION 24
GO
```

Drop the staging tables

Because all of the data is archived in the next (and final) step the staging data is not needed. A table drop is the f these tables.

```
DROP TABLE dbo.OrdersOctober2002
GO
DROP TABLE dbo.OrdersOctober2004
GO
```

Backup the filegroup

The choice in what you backup as your last step is based on your backup strategy. If a file- or filegroup-based bac chosen then a file or filegroup backup should be performed. If a full database based backup strategy has been cho backup or a differential backup can be performed.

```
BACKUP DATABASE SalesDB
      FILEGROUP = 'FG1'
TO DISK = 'C:\SalesDB\SalesDB.bak'
GO
```

List Partitioning – Regional Data

If your table has data from multiple regions and analysis often occurs within one region or you receive data for ea consider using defined "range" partitions in the form of a list. In other words, you will use a function that explicitly a value for a region. For example, consider a Spanish company that has clients in Spain, France, Germany, Italy, a data is always analyzed by country. For their table they can exactly five partitions, one for each country.

The creation of this list partition is almost identical to the range partition for dates except that the range's bounda values outside of the actual partition key. Instead, it is a list – not ranges. Although a list, the boundary conditions extreme left and the extreme right. To create 5 partitions you should specify only 4 in the partition function. The v ordered but the easiest way to get the correct number of partitions is to order the partition values and leave off th

```
ALTER DATABASE SalesDB
ADD FILEGROUP [Spain]
GO

ALTER DATABASE SalesDB
ADD FILE
      (NAME = N'SalesDBSpain',
        FILENAME = N'C:\SalesDB\SalesDBSpain.ndf',
        SIZE = 1MB,
        MAXSIZE = 100MB,
        FILEGROWTH = 5MB)
TO FILEGROUP [Spain]
GO
```

The next step is to create the function. The function will specify only 4 partitions using LEFT for the boundary conc
will include all countries except for the UK, as it is the last alphabetically in this list.

```
CREATE PARTITION FUNCTION CustomersCountryPFN(char(7))
AS
RANGE LEFT FOR VALUES ('France', 'Germany', 'Italy', 'Spain')
GO
```

To put the data on the filegroup for which it was named the partition scheme will be also be listed in alphabetical c
must be specified within the partitioning scheme's syntax.

```
CREATE PARTITION SCHEME [CustomersCountryPScheme]
AS
PARTITION CustomersCountryPFN
      TO ([France], [Germany], [Italy], [Spain], [UK])
GO
```

Finally, the customers table can be created on the new CustomersCountryPScheme.

```
CREATE TABLE [dbo].[Customers](
      [CustomerID] [nchar](5) NOT NULL,
      [CompanyName] [nvarchar](40) NOT NULL,
      [ContactName] [nvarchar](30) NULL,
      [ContactTitle] [nvarchar](30) NULL,
      [Address] [nvarchar](60) NULL,
      [City] [nvarchar](15) NULL,
      [Region] [nvarchar](15) NULL,
      [PostalCode] [nvarchar](10) NULL,
      [Country] [char](7) NOT NULL,
      [Phone] [nvarchar](24) NULL,
      [Fax] [nvarchar](24) NULL
) ON CustomersCountryPScheme (Country)
GO
```

So while range partitions are defined as supporting only "ranges" they can offer a way to perform other types of p

groupings (ranges or lists) and larger queries must analyze that data within these predefined and consistent range incoming and outgoing data within the same predefined ranges, then range partitioning is a simple choice. If you a over large amounts of data with no specific range to use or if all queries access most, if not all of the data, then us without any specific placement techniques is an easier to manage solution that will still yield performance gains.

Scripts from this Whitepaper:

The scripts used in the code samples for this whitepaper can be found in the SQLServer2005PartitionedTables.zip f is described below:

RangeCaseStudyScript1-Filegroups.sql – Includes the syntax to create the filegroups and files needed for the case study. This script allows modifications that will allow you to create this sample on a smaller set of disks with s instead of GB). It also has code to import data through INSERT…SELECT statements so that you can evaluate whe through the appropriate partitioning functions.

RangeCaseStudyScript2-PartitionedTable.sql – Includes the syntax to create the partition function, the partit partitioned tables related to the range partitioned table case study. This script also includes the appropriate constr

RangeCaseStudyScript3-JoiningAlignedTables.sql – Includes the queries demonstrating the various join strat for partitioned tables.

RangeCaseStudyScript4-SlidingWindow.sql – Includes the syntax and process associated with the monthly m partitioned table case study. In this script, you will "slide" data both in and out of the Orders table. Optionally, on same process to move data in and out for the OrderDetails table. Hint: see the Insert used in RangeCaseStudyScr correct columns of data to insert for OrderDetails.

RegionalRangeCaseStudyFilegroups.sql – Includes the syntax to create the filegroups and files needed for the table case study. In fact, this is a range partition to simulate a list partition scheme.

RegionalRangeCaseStudyPartitionedTable.sql – Includes the syntax to create the partition function, the parti regionally partitioned tables related to the range partitioned table case study.