



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 performance in larger databases, Microsoft SQL Server 2005 has new features that simplify the design. This whitepaper describes the evolution from manually partitioning data by creating your own tables to the preliminary features, which enabled partitioning in SQL Server 7.0 and SQL Server 2000, to the true partitioned table features available in SQL Server 2005. In SQL Server 2005, data-based partitioning features significantly simplify design and administration of partitioned tables while continuing to provide the flexibility and performance of manually partitioned tables. The paper's primary focus is to detail and document partitioning within SQL Server 2005 – giving you an understanding of how to use partitioned tables for the greatest benefit in your VLDB (Very Large Database). Although primarily a VLDB design tool, all databases start out large. SQL Server 2005 provides flexibility and performance while significantly simplifying the maintenance of partitioned tables. Review this document to get detailed information about why you should consider partitioning, the benefits 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 are available in the code samples for this whitepaper.

the [SQLServer2005Partitioning](#)

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 understand 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 : especially when large sets of data are being used in different ways maintenance operations can end up routinely n Performing maintenance operations on data which does not truly need it – is costly. The costs can be seen in perf 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 throug 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 acces 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 define (SalesJanuary2003, SalesFebruary2003, etc...) and then views for each quarter as well as a View for the entire year.

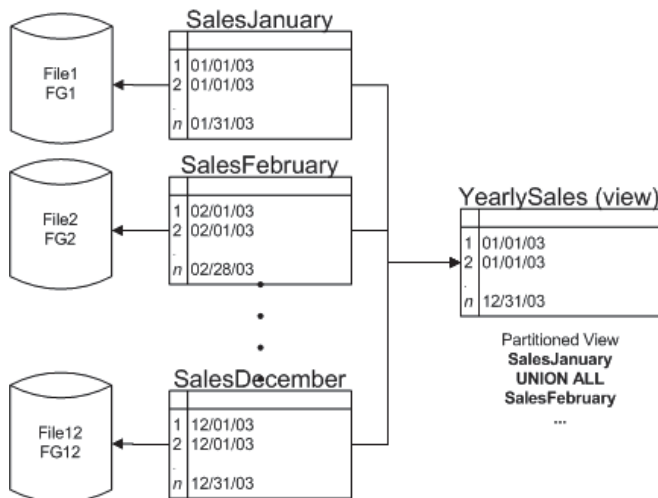


Figure 1: Partitioned Views in SQL Server 7.0/2000

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

```
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 set (the column on which the constraint was defined) then SQL Server will access only the necessary base table. If 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 taken as data can be verified against the constraint. Once the verification validates the existing data, the constraint is added. If the constraint is removed further inserts, updates, and deletes must adhere to the constraint in effect. Using this procedure to create constraints, developers can significantly reduce the complexity of their design using views without having to know the base table in which they were interested. With trusted constraints, SQL Server improves performance by removing the table from the execution plan.

Note: A constraint can become "untrusted" in various ways; for instance by performing a bulk-insert and not specifying the CHECK_CONSTRAINTS argument. Once a constraint has become untrusted, the query processor will revert to scanning the entire table 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 not support data modification statements; INSERT, UPDATE and DELETE statements were supported only against the base tables. SQL Server 2000 allows data modification statements to also benefit from the capabilities of SQL Server 7.0. By allowing data modification statements to use the same partitioned view structure as SELECT statements, modifications to the appropriate base table through the view. In order to configure this properly there are additional considerations for partitioning key and its creation; however, the basic principals are the same in that not only will SELECT queries benefit from the partitioning but the modifications will be as well. For details on the restrictions, setup, configuration and performance of 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

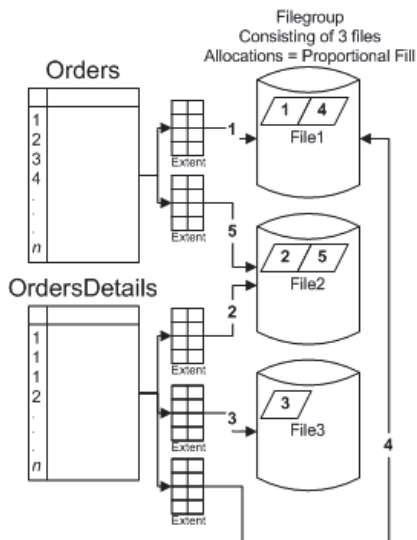
While the improvements in SQL Server 7.0 and SQL Server 2000 significantly enhanced performance using partitioned tables, the administration, design or development of a partitioned data set. Using partitioned views, all of the base tables (one view is defined) must be created and managed individually.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

For example, take a current month of OLTP (Online Transaction Processing) data that needs to be moved into an archive table. The latter table (to be used for read-only queries) is a single table with one clustered index and two non-clustered indexes. Loading 1GB (into the already indexed and active single table) creates blocking with current users as the table and its indexes are locked. Additionally, the loading process will take a significant amount of time as the table and its indexes are maintained as each row comes in. There are ways to speed up the bulk load; however, these can directly affect all users by sacrificing concurrency for speed. If this data were isolated into a newly-created (empty) table the load could occur in parallel (and index creation is possible) and then the indexes could be created after load (and index creation can be parallelized). Often you can achieve 10 times or better by using this scheme. In fact, by loading into an unindexed [heap] table you can take advantage of multiple data files in parallel or loading multiple chunks from the same file (defined by starting and ending row positions). Partitioning gives you this more granular control and does not restrict you to having all of the data in one location. A functional partitioning strategy in previous releases by dynamically creating and dropping tables and modifying the UNION view. However; in SQL Server 7.0 is more elegant; you can simply "switch in" the newly-filled partition(s) as an extra partition of the existing partitioned table (any old partition(s)). From end to end, the complete process takes only a short period of time and can be further improved by bulk loading and parallel index creation. More importantly, the partition is manipulated outside of the scope of the actual table until the partition is added. The result is that adding a partition typically occurs within only a few seconds.

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

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

In Figure 2, there are three files in a single filegroup. Two tables: Orders and OrderDetails have been placed on the filegroup. SQL Server proportionally fills the files within the filegroup by taking extent (6 or eight 8K Pages) allocations from each of the files as space is needed for the objects within the filegroups. When the OrderDetails table is created, the filegroup is empty. When an order comes in data is input into the Orders table and one row per line item is input into the OrderDetails table. SQL Server allocates an extent to the Orders table from File1 and another extent to the OrderDetails table from File2. It is likely that the OrderDetails table will grow more quickly and the next few allocations will go to the next table needing space. As OrderDetails grows, it will get the next extent from File3. SQL Server continues to "round robin" through the files within the filegroup. In the diagram below, follow each extent allocation from each extent to the appropriate filegroup - the extents are allocated as space is needed and each is numbered sequentially.



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 partitioning key and the scheme defines the location. In the case of a LEFT partition function, the first value will act as a lower boundary in the second partition. In the case of a RIGHT partition function, the first value will act as a lower boundary in the second partition. In the case of a RANGE partition function, the first value will act as a lower boundary in the second partition. In the case of a RANGE partition function, the first value will act as a lower boundary in the second partition. In the case of a RANGE partition function, the first value will act as a lower boundary in the second partition. Once the function is defined, a partition scheme defines 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 grouped together. By aligning rows with the same partitioning key from multiple tables to exist in filegroups on the same disks, SQL Server can, if the optimizer chooses, work with only the necessary groups of data (from each of the tables). 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 are equivalent 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, and
- 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 on 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 the partitioning column. This becomes important in the context of a query, subquery or another similar construct where the partitioning column may occur. Collocation is valuable because queries that join tables on the partitioning columns are in general much faster. In the context of the Orders and OrderDetails tables described above. Instead of filling the files proportionally, you can create a partitioning scheme. When defining the Orders and OrderDetails tables you will define them to use that same scheme. Having the same value for the partitioning key will be placed within the same file thereby isolating the necessary data. Related rows from multiple tables are partitioned in the same manner, SQL Server can join the partitions without having to scan an entire table or multiple partitions (if the table were using a different partitioning function) for matching rows. If the tables are not only aligned (because they use the same key) but they are said to be storage aligned because the same data is stored in the same files.

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

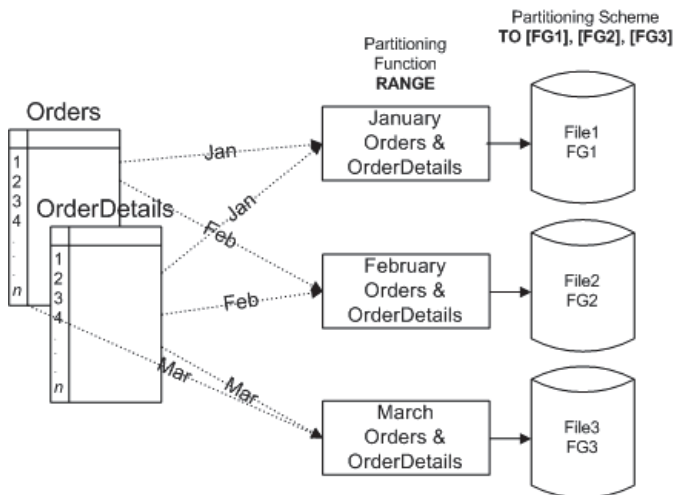


Figure 3: Storage Aligned Tables

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

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

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

Definitions and Terminology

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

Range Partitions

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

The primary use of range partitions is for data archiving, decision support (when often only specific ranges of data are needed, for example, only a given month or quarter) and for combined OLTP and DSS (Decision Support Systems) where data is updated frequently. 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 data can be added extremely quickly. Range partitions are best suited when data access is typically for decision support over large ranges of data. If you care where the data is specifically located so that only the appropriate partitions are accessed when necessary, range partitions are a good choice. When transactional data becomes available you will want to add that data in – easily and quickly. Range partitions are defined as you will need to define the boundary conditions for each of the partitions. Additionally, you will create a partitioning scheme that maps the partitions to a filegroup(s). However, they often have a consistent pattern to them so once defined they will likely be created programmatically (see Figure 4).

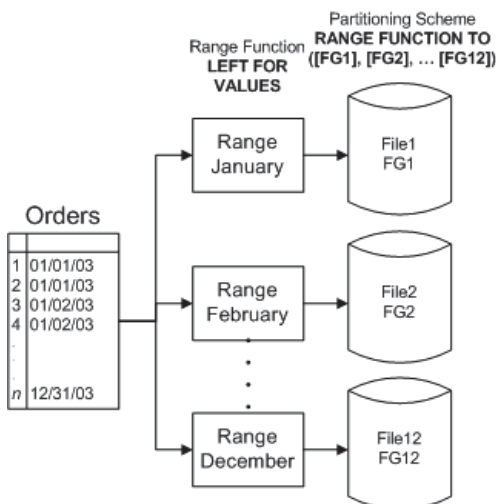


Figure 4: Range Partitioned Table - 12 Partitions.

Defining the Partitioning Key

The first step in partitioning tables and indexes is to define the data on which the partition is "keyed." The Partitioning Key is a column in the table and must meet certain criteria. The partition function defines the data type on which the logic is based. The physical placement of data is determined by the partition scheme. In other words, the scheme maps the data to specific file(s) and therefore disks. The scheme always uses a function to do this – if the function is a range function then the scheme must use filegroups. The filegroups do not need to be different however; you will use the same filegroup when you have multiple disks and preferably multiple CPUs. When the scheme is used with a table, you will define the partitioning key 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 not be contiguous. However, the usage of the data dictates a range partition because the pattern in which this table is used defines the partitioning key.

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 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 aligns the index's partitioning scheme unless the index is explicitly partitioned differently. When a table and its indexes are aligned, SQL Server can move partitions in and out of partitioned tables more effectively as all of the related data and indexes are divided by the same algorithm.

When the tables and indexes are defined with not only the same partition function but also the same partition scheme, the storage is aligned. In general, having the tables and indexes on the same file or filegroup can often benefit where you parallelize an operation across partitions. In the case of storage alignment and multiple CPUs, SQL Server can have data directly on a specific file or filegroup – and know that there are no conflicts in data access because all data requiring an 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 Switch

To aid in the usage of partitioned tables are several new concepts related to partition management. Because partitioned tables that need to scale to support better throughput, it is possible that the number of partitions chosen when the table was created will need to change over time. Using the ALTER TABLE statement with the new "split" option, you can add a new partition to a table. When a partition is "split," data may be moved to the new partition; however, for best performance rows should be moved in a single operation. A 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 partitioning, you make by stating which boundary point should be removed. Where only a specific period of data is needed and data is added on a regular basis (for example, monthly) you might want to archive one partition of data (the earliest month) after a new partition is added. For example, you might choose to have one year of data available and each month-end you will switch out the earliest month thereby differentiating between the current month's read/write OLTP data and the historical only analysis type of data. However, before you merge a boundary point you should switch out its associated data because this 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 September 2003 to the last month – August 2004. The current month of September 2004 is in another database, optimized for OLTP performance. In the current version of the table, there are 13 partitions. Twelve partitions which contain data (September 2003 through August 2004) and one partition which is empty (remember that a range partition includes the extreme left as well as the extreme right). To maintain data integrity by using a check constraint to require that the OrderDate be on or after September 1, 2003 and before September 1, 2005, the constraint will effectively keep the last partition empty.

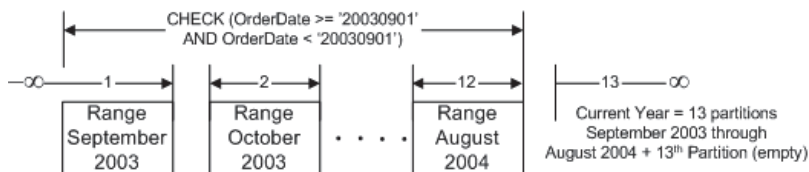


Figure 5: Range Partition Boundaries – Before Data Load/Archive

When October begins (in the OLTP database), we are ready to move September's data to the partitioned table. The process of switching the tables in and out is a very fast process and the preparation work can be performed outside the OLTP database. This scenario is explained in depth in the case study coming up but the idea is that you will use "staging tables" to move data between 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 filegroup** (and this is critical for success), SQL Server can make a metadata change. Switching out a partition can occur within seconds as SQL Server solely has to make the metadata change as opposed to running a delete that might take hours and create blocking in large tables. Once this partition is switched out, the table will still have 13 partitions; the first (oldest) partition is now empty and the last (most recent, also empty) partition is now the first.

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

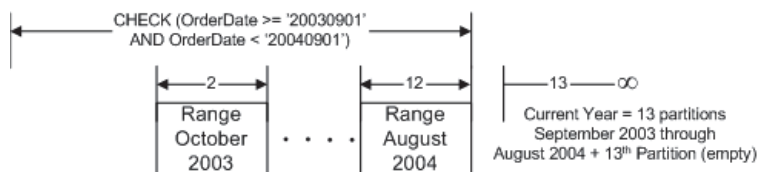


Figure 7: Merging a Partition

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 the partitioned table. To switch in the partition, you will first split the last (most recent, empty) range into two partitions. You need to update the table's constraint to allow the new range. Once again, the partitioned table will have 13 partitions in the sliding window scenario with a LEFT partition function will always remain empty.

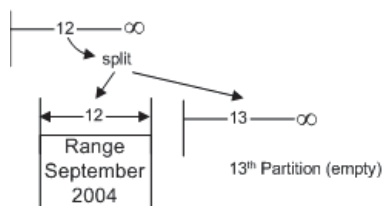


Figure 8: Splitting a Partition

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

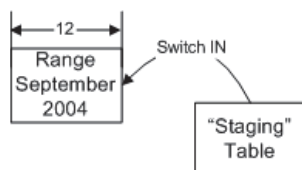


Figure 9: Switching a partition IN

The result of the table is:

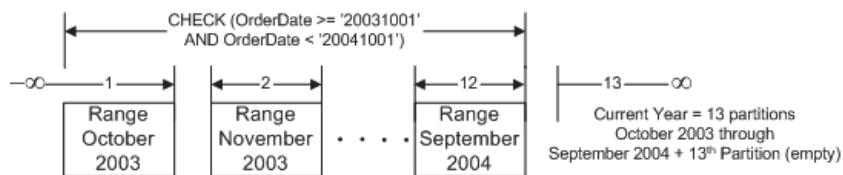


Figure 10: Range Partition Boundaries – After Data Load/Archive

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

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

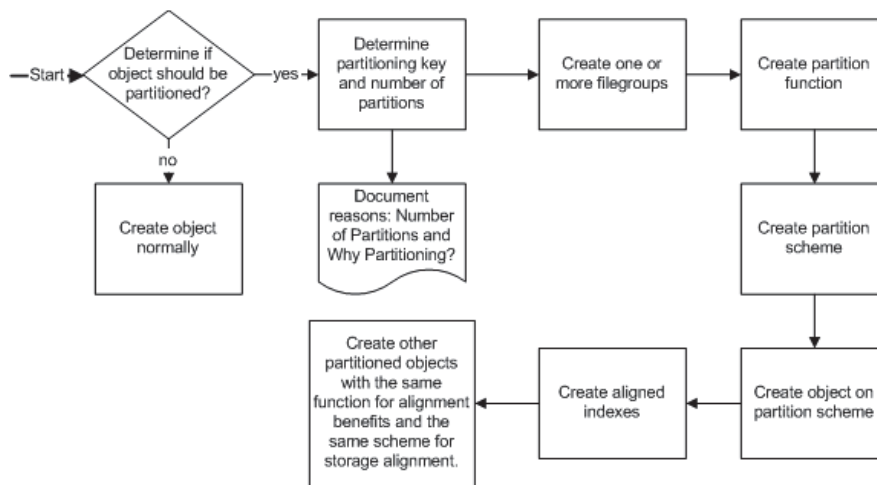


Figure 11: Steps to Create a Partitioned Table or Index

Determine IF Object should be partitioned

As described above this is the first and most important step. Not every table should be partitioned. Although partitioning adds administrative overhead and complexity to the implementation of your objects. Very great benefits, you may not want to consider it when your table is small. So what constitutes small v. large? Your size and maintenance requirements as well as whether or not your tables perform acceptably already. These are all factors that should be considered. 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 patterns, you should consider partitioning. If your data has logical groupings – and you often work with only a few of these defined groups, you should consider partitioning. 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 number of filegroups is partially dictated by hardware resources; generally you want to have the same number of filegroups that you have processors. These filegroups will reside on different disks. However, this primarily only pertains to systems where analysis tends to be on the entire data set. When you have multiple CPUs, SQL Server can process multiple partitions in parallel and therefore the overall processing time of large complex reports and analysis. In this case, you can have the benefit of parallel processing by 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 partitions – such as for more granular backup control – you should design your partitioned tables wisely so the grouped data resides on the same filegroup. Using ALTER DATABASE, you add a logical filegroup name – to which you add 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 cannot be created on filegroups without partitioning that table. To create a table on a single filegroup you use the ON clause of CREATE TABLE. To create a partitioned table you must first have a logical mechanism for the partitioning. Even though you may think of partitioning as logical partitions with a single table in mind, SQL Server allows you to reuse the partition definition for other objects. The concept of a partition from a table you create and name a partition structure through a partition function. Therefore, the first step in 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 partition must include all values even though the table may be (*and should be*) more restricted through a CHECK constraint. No part of the range – can be eliminated. Additionally, since data is likely to be switched in and out of the partition you will create an empty partition – in which you can repeatedly split – in order to make room for a new partition of data to "switch" in. This partition will always remain empty, as it will always be waiting to include future rows – which you will switch in periodically.

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

```
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 datetime data with 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 datetime data 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 best to use RIGHT (of the second partition set) with RIGHT and ending values (of the first partition set) with LEFT. The three following examples show 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 the partitioning correctly. Notice the simplicity with RIGHT – this is easier as the default time is 12:00:00.000am. For LEFT the added complexity is the precision of the datetime datatype. The reason that 23:59:59.997 MUST be chosen is that datetime data does not store the millisecond. Instead, datetime data is precise within 3.33 milliseconds. In the case of 23:59:59.999, this value is not available and instead the value is rounded to the nearest available value, 23:59:59.997.

To store one fourth of the Orders data in the four active partitions (each will represent one calendar quarter) and for 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 when defined as LEFT and one empty partition to the extreme LEFT when the partition function is defined as RIGHT. Review 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 23:59:59.997'

Boundary point '20010331 23:59:59.997':

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

Boundary point '20010630 23:59:59.997':

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

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 range partition function, the last value will define the last absolute partition value but there must be a place for all values – meaning that a partition must be created for the data rows greater than this value. There will always be one additional partition at the end for LEFT range partitions and one 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 a filegroup. When you define a partition scheme, you must make sure to name a filegroup for EVERY partition, even if multiple partitions use the same filegroup. For the range partition created previously (OrderDateRangePFN) there are 5 partitions, the last one must be created in the PRIMARY filegroup. There is no need for a special location for this partition because it will never have data.

```
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 table can be created. The table defines which "scheme" should be used and the scheme defines the function. To tie them together you must specify the column on which the partitioning function should apply. Range partitions always map to exact values and it should match the datatype of the boundary conditions defined within the partition function. Additionally, you can specifically restrict the data set to a more limited set (rather than from -infinity to positive infinity) then a check constraint can be used 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 column. The index is aligned with the table. It is not required, but often desired to have the table and its indexes aligned. Aligning allows for easier management and administration especially when working with the sliding window scenario. However, 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 ensure appropriate partition must be verified to guarantee uniqueness. Therefore if you need to partition a table on one column and create a unique index on a different column then they cannot be aligned; the index may be partitioned on the unique column or not partitioned at all. Be aware that this index will be 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 performance by loading 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 existing 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 loaded.

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 picture. However, for each step, there are specific settings and options that are available and in certain cases, a variety of options 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 used for Analysis. Analysis is often done for monthly, quarterly and/or yearly ranges of data. Because different analysts may want to look at different data at the same time – partitioning will better isolate their activity. In this scenario, the active data comes from multiple locations and is delivered in two standard format ASCII files. All files are placed on a centrally located fileserver on the first day of each month. Each file ranges in size but averages roughly 86,000 sales (Orders) per month. Each order therefore the OrderDetails files averages 226,180 rows. Each month roughly 25 million new Orders and 64 million rows are added and the historical analysis server maintains two year's worth of data active for analysis. Two year's worth of data is 172 million Orders and just over 1.5 billion OrderDetails rows. Because analysis is often done comparing months within the 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 to 3 months of data or up to 3 months of the current and past year (for example January through March 2003 with January through March 2002).

Orders Table, 24 total partitions, 25 filegroups, 12 Logical Drives (E:\ through P:\)

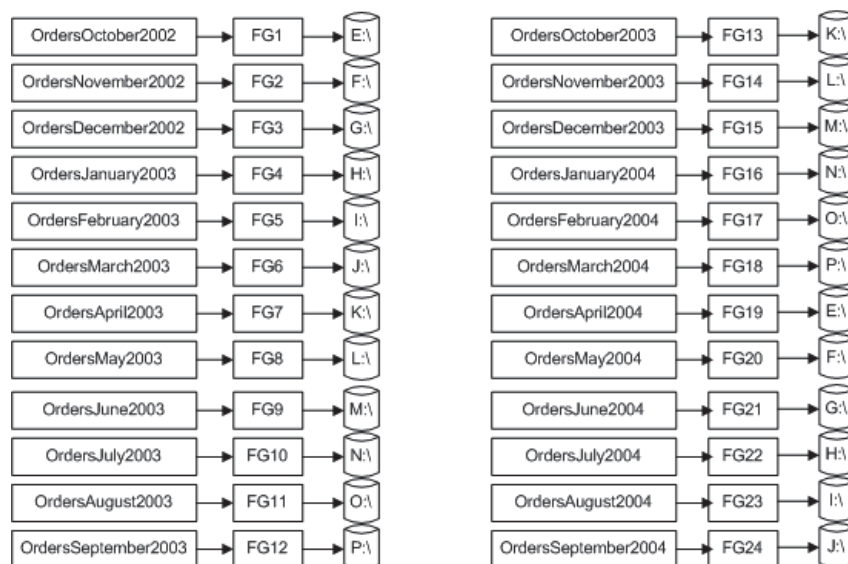


Figure 12: Partitioned Table Orders

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 database smaller tables such as Customers (9 million) and Products (386,750 rows), etc. Both the Orders and the OrderDetails same boundary conditions and the same placement on disk; in fact, they will use the same partitioning scheme. The two logical drives [Drive E:\ and F:\] in Figure 13) is that the data for Orders and OrderDetails will reside on the same months:

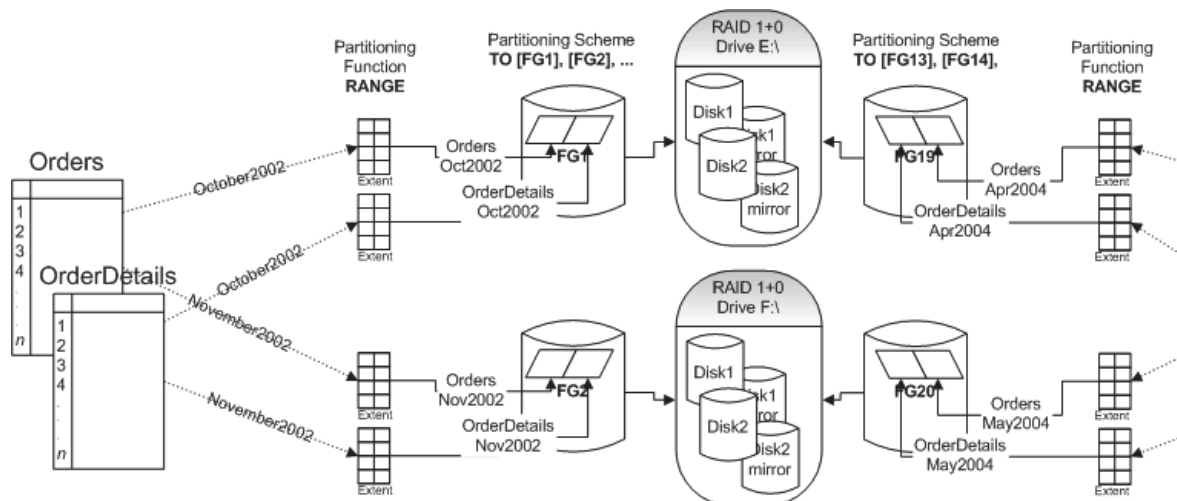


Figure 13: Range Partition to Extent Placement on Disk Arrays

While seemingly complex, this is quite simple to create. The hardest part in the design of the partitioned table is the large number of sources – 283 stores must have a standard mechanism for delivery. On the central server, however, the Orders table and one OrderDetails table to define. In order to create both tables as partitioned tables we must first define the function and the partition scheme. A partition scheme defines the physical placement of partition to disk so the file will be 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 to create filegroups (note: you cannot run this script without the appropriate drive letters; however, the script includes a "set drive" command modified for simplified testing. You can change the drive letters/locations to a single drive to test and learn the syntax. You can adjust the file sizes to MB instead of GB and consider a smaller number for initial size, depending on available disk space 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 will be partitioned on this value. In effect, the partitioning key value (if both expect to be the same key value) will be duplicated information; however, it is necessary for the alignment benefits and (in most cases) to narrow column (the `datetime` data type is 8 bytes). As described in *CREATE PARTITION FUNCTION for a Range Partition*, 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 creates 25 partitions. The 25th partition will remain empty. No special filegroup is needed for this empty partition, as no data will reside in it. To ensure that no data resides within this 25th filegroup a constraint will restrict the table's data. To do this, an appropriate disks a partition scheme is used to map the partition to the filegroup. The partition scheme will use appropriate filegroups for each of the 24 filegroups that will contain data and will use the `PRIMARY` filegroup for the 25th and empty partition.

```

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-defined filegroup (to create a non-partitioned table) or using a scheme (to create a partitioned table). As for which is best (even if this table will be populated and how many partitions you are going to manipulate. Populating the clustered index on it is likely to provide better performance than loading into a clustered table. Additionally, on a multi-processor machine with multiple CPUs you can load data into the table in parallel and then build the indexes in parallel. For this example, we create the table normally and then load the existing data through INSERT...SELECT statements. To create the Orders 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 table is created with the 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 [OrderDetailsModifiedDateFLT]
    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 determine which the data will reside. The following query is helpful as it returns the following information about each partition: many rows exist within each partition, the minimum and maximum OrderDate. A partition that does not contain rows is returned by 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 between Orders and OrderDetails. In this case, the clustered index will be defined on the Primary Key as you will also identify both of these tables with the same partitioning scheme (and for OrderDetails you will add the LineNumber to the index for uniqueness). When created, the default behavior for partitioned tables is to align the index with the partitioned table on the same scheme; the scheme does not need to be explicitly defined.

```

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. In the first step, the partitions are joined first and then the subsets are added together. Moreover, regardless of how the partitions are joined, SQL Server 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. The query is designed to retrieve information from the third quarter only. Typically, the third quarter includes slower months for order processing; however, in this case, the third months were some of the largest months for Orders. In this case, we want to see if there are any interesting trends in the quantities ordered and their order dates – but only for the third quarter. To ensure that aligned partitioned tables benefit from partition elimination when joined, you must be sure to state each table's partitioning range. In this case, since the Orders table has a composite key of both OrderDate and OrderID the join between these tables will show that OrderDate must be equal to the OrderDate in the OrderDetails table. 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
FROM Orders o
JOIN OrderDetails od
    ON o.OrderDate = od.OrderDate
    AND o.OrderID = od.OrderID

```

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

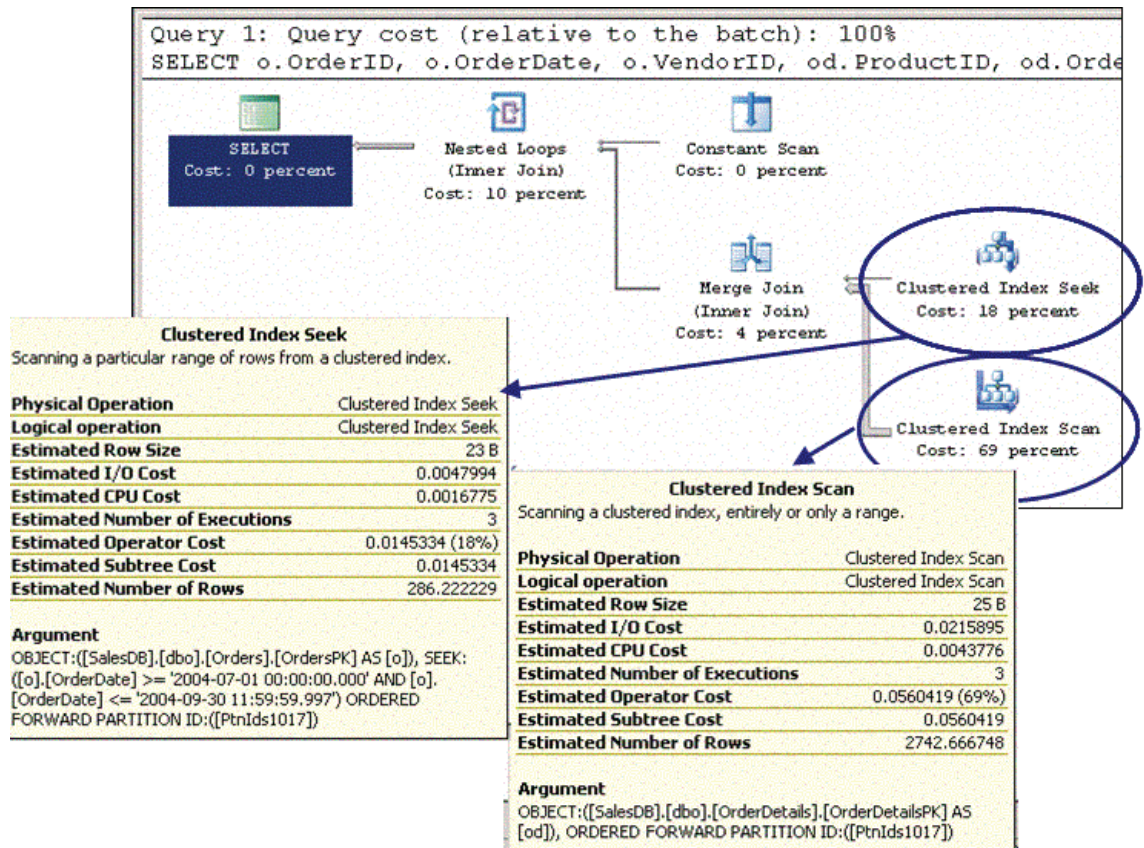


Figure 14: Number of Executes

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 expression comes from. If you hover over the Constant Scan at the top of the showplan you will see that it shows an Argument of **VALUES((21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495, 1496, 1497, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 1506, 1507, 1508, 1509, 1510, 1511, 1512, 1513, 1514, 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1522, 1523, 1524, 1525, 1526, 1527, 1528, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1546, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 1573, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607, 1608, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, 1668, 1669, 1670, 1671, 1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 1772, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 1801, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812, 1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 1848, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065**

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 `df` 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.

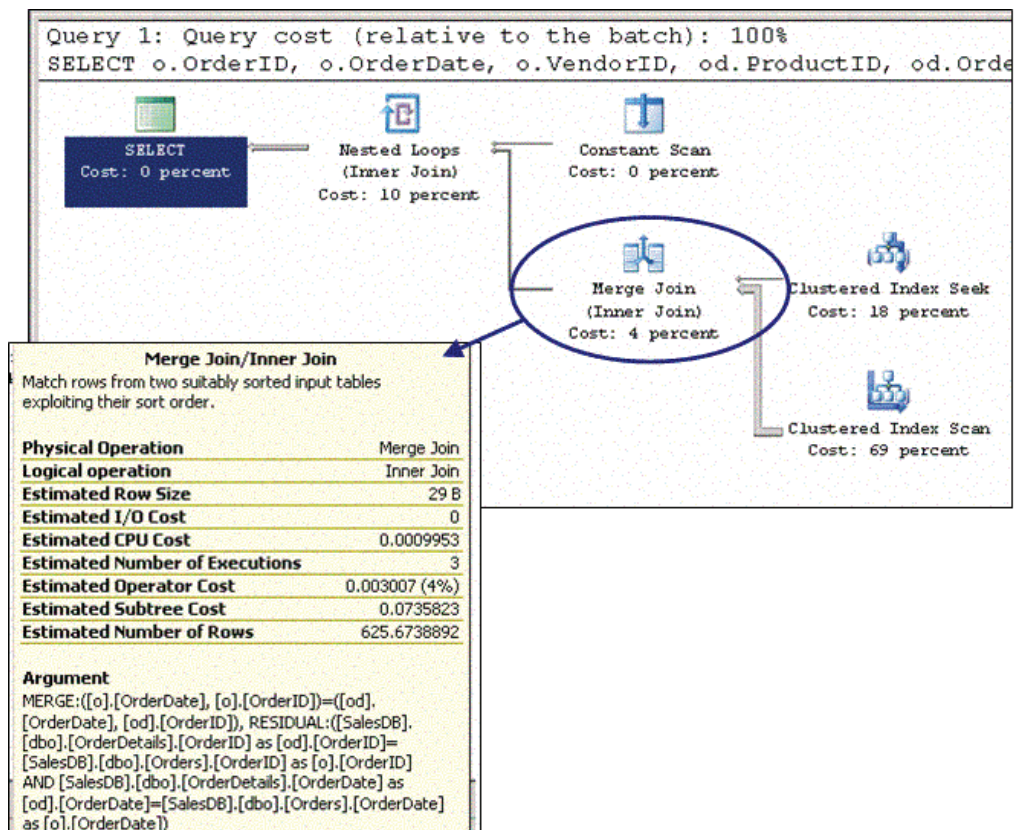


Figure 16: Joining Partitioned Tables

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 plan the same filegroups, FG1 in this case, then you will need to determine how to handle the space and archiving requirement. You 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 you have space to keep both the current and the month to be archived then you will need to switch the old partition out first.

Regardless, archiving should be easy – and is probably already being done. A good practice for archiving is to back up immediately after the new partition is loaded and switched in (not just before you plan to switch it out). The reason 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 structure. A database backup is not your only option however, there are numerous backup strategies which can be implemented many offer better granularities for backup and restore). For future backups because so much of the data is **not** changed the individual filegroups after they are loaded. In fact, this should become part of your rolling partition strategy. For more information see 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 of steps are complex however each month the overall process will be identical. All that will change are the dates. By using dynamic SQL you 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 contains code for 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 restrict its data to only valid data for the soon-to-be partition of the table. However, for better performance you want to load 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 month. Your process runs detecting which month is easy with built-in functions such as DATENAME(m, getdate()). The table is created as a new table – so the primary change for each month is to the table name. However, you could use the same name for the existing table – so the primary change for each month is to the table name. However, you could use the same name for the existing table – so the primary change for each month is to the table name. However, you could use the same name for the existing table – so the primary change for each month is to the table name.

```

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 split. To determine the appropriate filegroup with which to work, use the following system table query combined with the information shown earlier. Specify any date within the range being rolled out. This is the partition (and filegroup) in which the data will be 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 INSERT. In automation: this process is the most complex to automate. You will need to make sure that all files have been loaded and want to load them in parallel. To control this process you might consider a table that keeps track of which files have been loaded where the files are located. You can create a SQL Agent job that checks for files every few minutes, picks up the files and 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 be added with the WITH CHECK setting. The WITH CHECK setting is the default so it does not need to be specified but it is important NOT to say WITHOUT CHECK.

```

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 partition.

```

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 partition (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 longer the "next used" partition scheme. Since you will roll the new data through the same existing 24 partitions then you will need to use the "next 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 (OrderDate < '2004-11-01'). For 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 final step for 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 backup strategy is chosen then a file or filegroup backup should be performed. If a full database based backup strategy has been chosen then a full 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 each region, consider using defined “range” partitions in the form of a list. In other words, you will use a function that explicitly defines a value for a region. For example, consider a Spanish company that has clients in Spain, France, Germany, Italy, and the UK. The data is always analyzed by country. For their table they can create 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 boundary values are outside of the actual partition key. Instead, it is a list – not ranges. Although a list, the boundary conditions are at the extreme left and the extreme right. To create 5 partitions you should specify only 4 in the partition function. The values are ordered but the easiest way to get the correct number of partitions is to order the partition values and leave off the last value.

```

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 condition and 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 order and 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 have over large amounts of data with no specific range to use or if all queries access most, if not all of the data, then using 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](#) file as 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 100GB instead of 1TB. It also has code to import data through INSERT...SELECT statements so that you can evaluate what happens through the appropriate partitioning functions.

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

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

RangeCaseStudyScript4-SlidingWindow.sql – Includes the syntax and process associated with the monthly moving average partitioned table case study. In this script, you will “slide” data both in and out of the Orders table. Optionally, you can use the same process to move data in and out for the OrderDetails table. Hint: see the Insert used in RangeCaseStudyScript1 for correct columns of data to insert for OrderDetails.

RegionalRangeCaseStudyFilegroups.sql – Includes the syntax to create the filegroups and files needed for the regional range partitioned 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 partitioned tables related to the regional range partitioned table case study.