Microsoft SQL Server 2005

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 and manageability in larger databases, Microsoft SQL Server 2005 has new features that simplify the design. This whitepaper describes the logical progression from manually partitioning data by creating your own tables to the preliminary features, which enabled partitioning through views in SQL Server 7.0 and SQL Server 2000, to the true partitioned table features available in SQL Server 2005. In SQL Server 2005, the new table-based partitioning features significantly simplify design and administration of partitioned tables while continuing to improve performance. The paper's primary focus is to detail and document partitioning within SQL Server 2005 – giving you an understanding of why, when and how to use partitioned tables for the greatest benefit in your VLDB (Very Large Database). Although primarily a VLDB design strategy, not all databases start out large. SQL Server 2005 provides flexibility and performance while significantly simplifying the creation and maintenance of partitioned tables. Review this document to get detailed information about why you should consider partitioned tables, what 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 SQLServer2005PartitionedTables.zip file.

Table of Contents

Why do you need Partitioning?

Before one can talk about how to implement partitioning and the features of partitioning one must first understand the need; what are partitions and why might someone consider using them? When you create tables, you design those tables to store information about an entity – i.e. customers or sales. Each table should have attributes that describe only that entity and for customers and sales the historical premise is that all of your customers and all of your sales go into their respective tables. While a single table for each entity is the easiest to design and understand, it may not be the best for performance, scalability and manageability especially as the table grows large. Partitioning can provide benefits for both large tables (and/or their indexes) and tables which have varying access patterns. More specifically, through partitioning practices large tables have better scalability and manageability and the use of tables that have changing 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 measured in hundreds of gigabytes or even terabytes but the term does not necessarily specify individual table sizes. A large table is one that does not perform as desired or one where the maintenance costs have gone beyond pre-defined maintenance periods. Furthermore, a table can be considered large if one user's activities significantly affect another or if maintenance operations affect other user's abilities. In effect, this even limits availability. Even though the server is available, how can you consider your database available when the sales table's performance is severely degraded or even inaccessible during maintenance for 2 hours per day, per week, or even per month? In some cases, periodic 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 different usage patterns. Although usages patterns may not always vary (and this is not a requirement for partitioning), when usage patterns do vary there can be additional gains. Again, thinking in terms of sales, the current month's data is read-write while the previous month's data (and often the larger part of the table) is read-only. Large tables where the data usage varies or large tables where the maintenance overhead is overwhelming can limit the table's ability to respond to varied user requests, in turn limiting both availability and scalability. Moreover, especially when large sets of data are being used in different ways maintenance operations can end up routinely maintaining static data. Performing maintenance operations on data which does not truly need it – is costly. The costs can be seen in performance problems, blocking problems, backups (space, time and operational costs) as well as negatively impacting the overall scalability of the server.

Furthermore, if a large table exists on a system with multiple CPUs, partitioning the table can lead to better performance through parallel operations. Large-scale operations across extremely large data sets – typically many million rows – can benefit by performing multiple operations against individual subsets in parallel. A simple example of performance gains over partitions can be seen in previous releases with aggregations. For example, instead of aggregating a single large table, SQL Server can work on partitions independently and then aggregate the aggregates. In SQL Server 2005, joins can benefit directly from partitioning; SQL Server 2000 supported parallel join operations on subsets yet needed to create the subsets on the fly. In SQL Server 2005, related tables (i.e. Order and OrderDetails) that are partitioned to the same partitioning key and the same partitioning function are said to be aligned. When the optimizer detects that two partitioned and aligned tables are joined, SQL Server 2005 can choose to join the data which resides on the same partitions first and then 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 amounts of data into smaller more manageable chunks (i.e. partitions). The type of partitioning described in this paper is termed horizontal partitioning. With horizontal partitioning, large chunks of rows will be stored in multiple separate partitions. The definition of the partitioned set is customized, defined, and managed – by your needs. Partitioning in SQL Server 2005 allows you to partition your tables based on specific data usage patterns using defined ranges. Finally, SQL Server 2005 offers numerous options for the long-term management of partitioned tables and 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 release. However, without features to aid in creating and maintaining your partitioning scheme, partitioning has often been cumbersome and underutilized. Typically, the design is misunderstood by users and developers and the benefits are diminished. However, because of the significant performance gains inherent in the concept, SQL Server 7.0 began improving the features enabling forms of partitioning through partitioned views and SQL Server 2005 now offers the largest advances through partitioned tables.

Partitioning Objects manually in releases before SQL Server 7.0

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 coding and querying practices. By creating multiple tables and then managing access to the correct tables through stored procedures, views or client applications you could often improve performance for some operations but at the cost of complexity of design. Each user and developer needed to be aware of and properly reference the correct tables. Each partition was created and managed separately and views were used to simplify access; however, this solution yielded few performance gains. When a UNIONed view existed to simplify user/application access, the query processor had to access every underlying table in order to determine if data was needed for the result-set. If only a limited subset of those underlying tables was needed, then each user and developer needed to know the design in order to reference only the appropriate table(s).

Partitioned Views in SQL Server 7.0

The challenges faced by manually creating partitions in releases prior to SQL Server 7.0, were primarily related to performance. While views simplified application design, user access and query writing, they did not offer performance gains. With the release of SQL Server 7.0, views were combined with constraints to allow the query optimizer to remove irrelevant tables from the query plan (i.e. partition elimination) and significantly reduce overall plan cost when a UNIONed view accessed multiple tables.

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

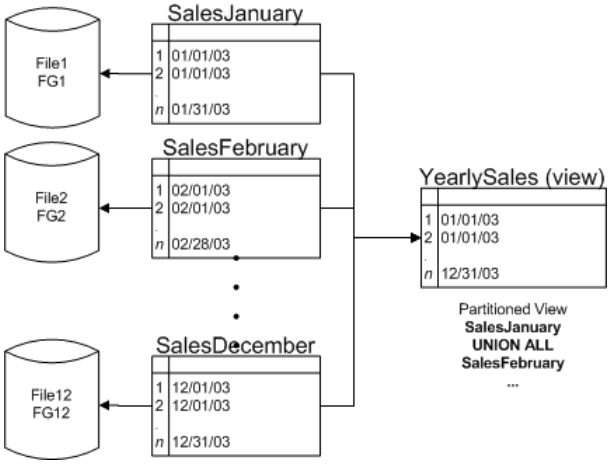

Figure 1: Partitioned Views in SQL Server 7.0/2000

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 results based on the partition key (the column on which the constraint was defined) then SQL Server will access only the necessary base table. "Trusted" constraints are constraints where SQL Server is able to guarantee that all data adheres to the properties defined by the constraint. When the constraint is created, the default behavior is to create the constraint WITH CHECK. This setting causes a schema lock to be taken on the table so that the data can be verified against the constraint. Once the verification validates the existing data, the constraint is added; once the schema lock is removed further inserts, updates, and deletes must adhere to the constraint in effect. Using this procedure to create "trusted" constraints, developers can significantly reduce the complexity of their design using views without having to know and/or directly access the table in which they were interested. With trusted constraints, SQL Server improves performance by removing the unnecessary tables 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 all base tables as it 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 yield any benefits for data modification statements; INSERT, UPDATE and DELETE statements were supported only against the base tables, not directly against the views which UNIONed the tables. SQL Server 2000 allows data modification statements to also benefit from the partitioned view capabilities of SQL Server 7.0. By allowing data modification statements to use the same partitioned view structure, you can direct modifications to the appropriate base table through the view. In order to configure this properly there are additional restrictions on the partitioning key and its creation; however, the basic principals are the same in that not only will SELECT queries be sent directly to the appropriate base tables but the modifications will be as well. For details on the restrictions, setup, configuration and best practices for 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 views, they did not simplify the administration, design or development of a partitioned data set. Using partitioned views, all of the base tables (on which the view is defined) must be created and managed individually. Application design is easier and users benefit by not needing to know which base table to directly access but administration is complex as there are numerous tables to manage and data integrity constraints must be managed for each table. Because of the management issues, partitioned views were often used to separate tables only when data needed to be "archived" or loaded. When data was moved into the read-only table or when data was deleted from the read-only table, the operations were expensive – taking time, log space and often creating blocking.

Additionally, because prior partitioning strategies required the developer to create individual tables and indexes and then UNION them through views, the optimizer was required to validate and determine plans for each partition (as indexes could have varied). Therefore, query optimization time in SQL Server 2000 often goes up linearly with the number of processed partitions. This is not in the case of partitioned tables in SQL Server 2005 where each partition has the same indexes by definition.

For example, take a current month of OLTP (Online Transaction Processing) data that needs to be moved into an analysis table at month end. The latter table (to be used for read-only queries) is a single table with one clustered index and two non-clustered indexes; the bulk load of 1GB (into the already indexed and active single table) creates blocking with current users as the table and/or indexes become fragmented and/or locked. Additionally, the loading process will take a significant amount of time as the table and indexes must be maintained as each row comes in. There are ways to speed up the bulk load; however, these can directly affect all other users thereby sacrificing concurrency for speed. If this data were isolated into a newly-created (empty) table the load could occur first (and parallel loads are possible) and then the indexes could be created after load (and index creation can be parallelized). Often you will realize gains of 10 times or better by using this scheme. In fact, by loading into an unindexed [heap] table you can take advantage of multiple CPUs by loading multiple data files in parallel or loading multiple chunks from the same file (defined by starting and ending row position). In any release, partitioning gives you this more granular control and does not restrict you to having all of the data in one location – with heavily fragmented indexes – and no real ability to control any aspect of it at a more granular level. A functional partitioning strategy could be achieved in previous releases by dynamically creating and dropping tables and modifying the UNION view. However; in SQL Server 2005 the solution is more elegant; you can simply "switch in" the newly-filled partition(s) as an extra partition of the existing partition scheme and "switch out" any old partition(s). From end to end, the complete process takes only a short period of time and can be further improved using parallel bulk loading and parallel index creation. More importantly, the partition is manipulated outside of the scope of the table so there is NO effect on 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 then when new data is ready to be migrated in (for example the current month) then the oldest data (maybe the parallel month from the previous year) can be removed. In this same example, you will likely see several orders of magnitude better performance improvement by utilizing partitioning. While this may seem extreme, consider the difference; when all of the data is in a single table the deleting 1GB of data (the oldest data), requires row-by-row manipulation of the table as well as its associated indexes. The process of deleting data creates a significant amount of log activity and does not allow log truncation for the length of the delete (remember the delete is a single auto-commit transaction; however, you can control the size of the transaction by performing multiple deletes – where possible) and which requires a [potentially much] larger log. To remove the same amount of data – by removing the specific partition from a partitioned table – all that must be done is a removal of the partition (which is a meta data operation) and then drop or truncate the standalone table.

Moreover, without knowing how to best design partitions one might not be aware that the use of filegroups in conjunction with partitions is ideal for implementing partitioning. Filegroups allow you to place individual tables on different physical disks. If a single table spans multiple files using filegroups then the actual physical location of data could not be predicted. For systems where parallelism is not expected, 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 this filegroup. When tables are placed on a filegroup SQL Server proportionally fills the files within the filegroup by taking extent (64KB chunks which equal eight 8K Pages) allocations from each of the files as space is needed for the objects within the filegroups. When the Orders and OrderDetails tables are created, the filegroup is empty. When an order comes in data is input into the Orders table (one row per order) and one row per line item is input into the OrderDetails table. SQL Server allocates an extent to the Orders table from File1 and then another extent to the OrderDetails table from File2. It is likely that the OrderDetails table will grow more quickly than the Orders table and the next few allocations will go to the next table needing space. As OrderDetails grows, it will get the next extent from File3 and SQL Server continues to "round robin" through the files within the filegroup. In the diagram below, follow each table to an extent and from each extent to the appropriate filegroup – the extents are allocated as space is needed and each is numbered based on flow.

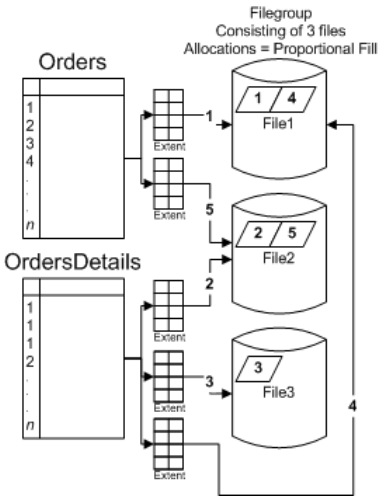

Figure 2: Proportional Fill using Filegroups

SQL Server will continue to balance allocations among all of the objects within that filegroup. While SQL Server benefits from using more disks for a given operation, it is not as optimal from a management or maintenance perspective or where usage patterns are very predictable (and isolated).

With partitioned tables in SQL Server 2005, a table can be designed (using a "function" and a "scheme") such that all rows that have the same partitioning key are placed directly on (and will always go to) a specific location. The function defines the partition boundaries and in which partition the first value should be placed. In the case of a LEFT partition function, the first value will act as an upper boundary in the first partition. In the case of a RIGHT partition function, the first value will act as a lower boundary in the second partition. You will see many more details regarding partition functions later within this paper. Once the function is defined, a partition scheme can be created to define the physical mapping of the partitions to their location within the database – based on a partition function. When multiple tables use the same function (but not necessarily the same scheme) rows that have the same partitioning key will be grouped similarly. This concept is called alignment. By aligning rows with the same partition key from multiple tables to exist in filegroups on the same or different physical disks, SQL Server can, if the optimizer chooses, work with only the necessary groups of data (from each of the tables). To achieve alignment two partitioned tables or indexes must have some correspondence between their respective partitions. They must use "equivalent" partition functions and some relation with respect to the partitioning columns. Two partition functions can be used to align data 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 collation if applicable)
- 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 if it is not partitioned 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 equi-join is on the partitioning column. This becomes important in the context of a query, subquery or another similar construct where equi-join predicates may occur. Collocation is valuable because queries that join tables on the partition columns are in general much faster. Take for example the Orders and OrderDetails tables described above. Instead of filling the files proportionally, you can create a partition scheme that maps to three filegroups. When defining the Orders and OrderDetails tables you will define them to use that same scheme. Related data (related by having the same value for the partition key) will be placed within the same file thereby isolating the necessary data for the join. When related rows from multiple tables are partitioned in the same manner, SQL Server can join the partitions without having to search through an entire table or multiple partitions (if the table were using a different partitioning function) for matching rows. In this case, the objects are not only aligned (because they use the same key) but they are said to be storage aligned because the same data resides within 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 end up on the same filegroup. When related data is aligned, SQL Server 2005 can effectively work on large sets in parallel. All of the sales data for January (for both the Orders and OrderDetails tables) is on the first filegroup and February data on the second filegroup and so forth.

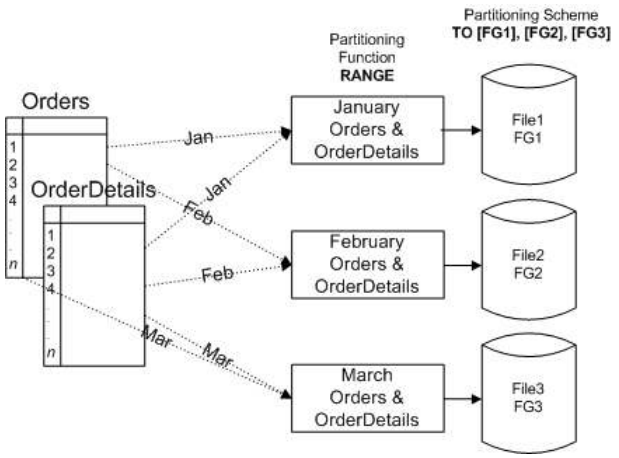

Figure 3: Storage Aligned Tables

SQL Server allows partitioning based on ranges. Tables as well as indexes can use the same scheme for better alignment. Good design 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 partitioned table were major 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 purposes
- Load data into a new partition of existing partitioned table with minimal disruption in data access in the remaining partitions
- Load data into a new partition of existing partitioned table with performance equal to loading the same data into a new empty table
- Archive and/or remove a portion of a partitioned table while minimally impacting access to the remainder of the table
- Allow partitions to be maintained by "switching" partitions in and out of the partitioned table
- Allow better scaling and parallelism for extremely large operations over multiple related tables
- Improve performance over all partitions
- Improve query optimization time because each partition does not need to be optimized separately

Definitions and Terminology

To implement partitions in SQL Server 2005 you must be familiar with a few new concepts, terms and syntax. In previous releases a table was always both a physical and a logical concept, yet with SQL Server 2005 Partitioned Tables and Indexes you have multiple choices for how and where you store a table. In SQL Server 2005, tables and indexes can be created with the same syntax as previous releases – as a single tabular structure that is placed on the DEFAULT filegroup or a user-defined filegroup. Additionally, in SQL Server 2005 table and indexes can also be created on a partitioning scheme. The partitioning scheme maps the object to a filegroup or possibly multiple filegroups. 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 from earlier 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 boundaries are chosen by the developer – and can be changed if data usage patterns change. Typically, these ranges are date based or based on some ordered groupings of data.

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

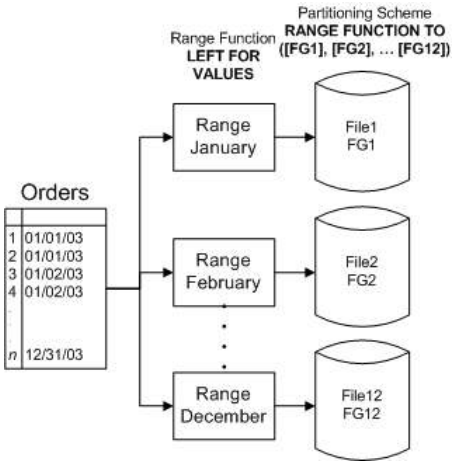

Figure 4: Range Partitioned Table - 12 Partitions.

Defining the Partitioning Key

The first step in partitioning tables and indexes is to define the data on which the partition is "keyed." The Partition Key must exist as a column in the table and must meet certain criteria. The partition function defines the data type on which the logical separation of data is based. The physical placement of data is determined by the partition scheme. In other words, the scheme maps the data to a filegroup(s) that maps the data to specific file(s) and therefore disks. The scheme always uses a function to do this – if the function defines five partitions then the scheme must use five filegroups. The filegroups do not need to be different however; you will get better performance when you have multiple disks and preferably multiple CPUs. When the scheme is used with a table, you will define the column that is used 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 truly balanced at all. However, the usage of the data dictates a range partition because the pattern in which this table is used defines specific boundaries of analysis (a.k.a. ranges). The partition key for a range function can consist of only one column and the partitioning function will include the entire domain – even when that data may not be possible within the table. In other words, boundaries are defined for each partition but the first partition and the last partition will [potentially] include rows for the extreme left (values less than the lowest boundary condition) and for the extreme right (values greater than the highest boundary condition). Partitions must be combined with CHECK constraints to enforce your business rules and data integrity constraints (i.e. restrict the data set to a finite range rather than infinite range). Range partitions are ideal when maintenance and administration requires archiving of large ranges of data on a regular basis and when queries access a large amount of data – but only within a subset of the ranges.

Index Partitioning

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

When the tables and indexes are defined with not only the same partition function but also the same partition scheme then they are said to be storage aligned. In general, having the tables and indexes on the same file or filegroup can often benefit where multiple CPUs can parallelize an operation across partitions. In the case of storage alignment and multiple CPUs, SQL Server can have each processor work directly on a specific file or filegroup – and know that there are no conflicts in data access because all data required for a particular join or 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 partitions are used for large tables that need to scale to support better throughput, it is possible that the number of partitions chosen when the partition function was created will need to change over time. Using the ALTER TABLE statement with the new "split" option, you can add another partition to the table. When a partition is "split," data may be moved to the new partition; however, for best performance rows should not move. A full 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 partitions, a merge request is made by stating which boundary point should be removed. Where only a specific period of data is needed and data archiving is occurring on a regular basis (for example, monthly) you might want to archive one partition of data (the earliest month) after adding the newest partition in. For example, you might choose to have one year of data available and each month-end you will switch in the current month and then switch

out the earliest month thereby differentiating between the current month's read/write OLTP data and the ealier month's read-only analysis type of data. However, before you merge a boundary point you should switch out its associated data otherwise the merge could be an expensive operation. There is a specific flow to this process to make it the most efficient. The concept of switch, merge and split 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 through to last month – August 2004. The current month of September 2004 is in another database, optimized for OLTP performance. In the read-only version of the table, there are 13 partitions. Twelve partitions which contain data (September 2003 through August 2004) and one final partition which is empty (remember that a range partition includes the extreme left as well as the extreme right). The table might enforce data integrity by using a check constraint to require that the OrderDate be on or after September 1, 2003 and before September 1, 2004; the constraint will effectively keep the last partition empty.

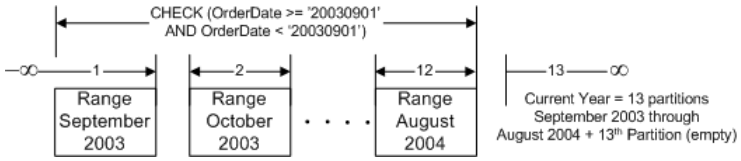

Figure 5: Range Partition Boundaries – Before Data Load/Archive

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

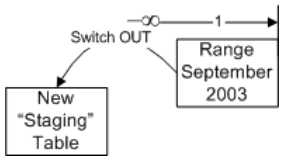

Figure 6: Switching a partition OUT

To remove the first (oldest) partition (September 2003) you will use the new "merge" option (Figure 7) with ALTER TABLE. By merging a 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; in this case, because the first partition is empty no rows need to move (from the first to the second partition). If the first partition is NOT empty and the boundary point is "merged" then rows will have to move from the first to the second partition and this could be a very expensive 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.

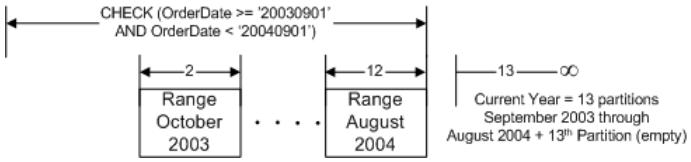

Figure 7: Merging a Partition

Finally, the new table has to be switched in to the partitioned table. Remember, in order for this to be performed as nothing more than a metadata change, all of the work to load and build indexes has to have already occurred on a new table – outside of the bounds of the partitioned table. To switch in the partition, you will first split the last (most recent, empty) range into two partitions. Additionally, you will need to update the table's constraint to allow the new range. Once again, the partitioned table will have 13 partitions. The last partition in the sliding window scenario with a LEFT partition function will always remain empty.

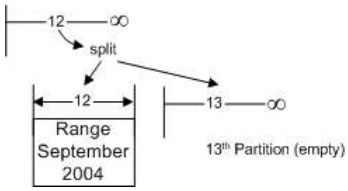

Figure 8: Splitting a Partition

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

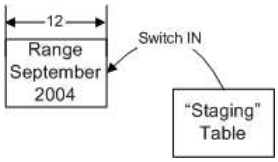

Figure 9: Switching a partition IN

The result of the table is:

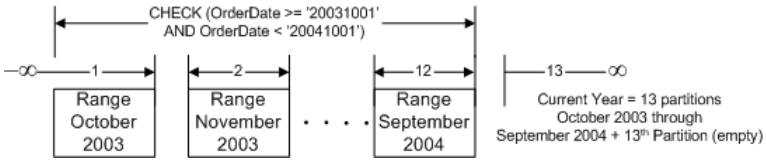

Figure 10: Range Partition Boundaries – After Data Load/Archive

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

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

Steps for Creating Partitioned Tables

Now that you have a logical understanding of why partitioned tables exist and what they offer, it is good to get a logical flow (Figure 11) of the process of creating a partitioned table. Many features assist in creating and defining a partitioned table. The logical flow is as follows:

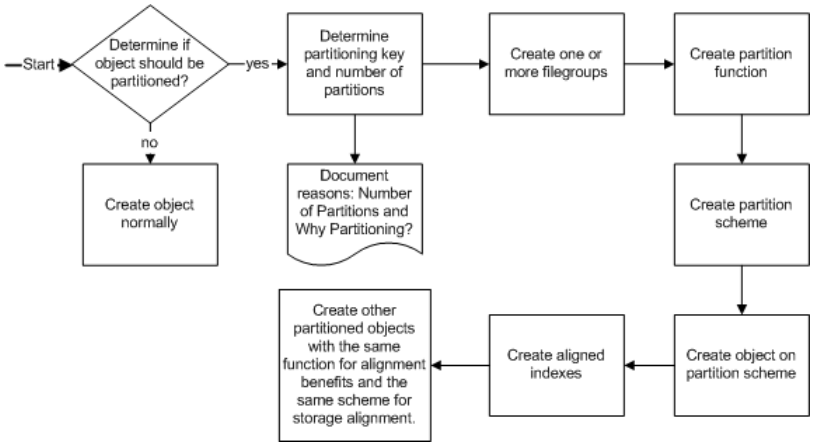

Figure 11: Steps to Create a Partitioned Table or Index

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 is transparent to the application, partitioning adds administrative overhead and complexity to the implementation of your objects. While partitioning can offer great benefits, you may not want to consider it when your table is small. So what constitutes small v. large? Your performance requirements and maintenance requirements as well as whether or not your tables perform acceptably already. These are all factors in determining whether an existing table should be partitioned. For example, archiving a range of data out of a single sales table takes a significant amount of time – minutes (or more) to add new rows in (via INSERT…SELECT) and minutes (or more) to remove old data (via DELETE…WHERE). If these create an unnecessary burden on system performance and resources then you should consider partitioning, otherwise it might form 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, use the range partitioning mechanism. If your data has logical groupings – and you often work with only a few of these defined subsets at a time then 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 partitions and preferably, these filegroups will reside on different disks. However, this primarily only pertains to systems where analysis tends to be performed over the entire data set. When you have multiple CPUs, SQL Server can process multiple partitions in parallel and therefore significantly reduce the overall processing time of large complex reports and analysis. In this case, you can have the benefit of parallel processing as well as 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(s). Filegroups can consist of one or more files and each partition must map to a filegroup. A single filegroup can be used for multiple partitions but for better data management – such as for more granular backup control – you should design your partitioned tables wisely so that only related or logically grouped data resides on the same filegroup. Using ALTER DATABASE, you add a logical filegroup name – to which files can be added. To 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
ADD FILE
   (NAME = N'2003Q3',
   FILENAME = N'C:\AdventureWorks\2003Q3.ndf',
   SIZE = 5MB,
   MAXSIZE = 100MB,
   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 multiple 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 and define the logical partitions with a single table in mind, SQL Server allows you to reuse the partition definition for other objects. To separate 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 function must include all values even though the table may be *(and should be)* more restricted through a CHECK constraint. No values – from either end of the range – can be eliminated. Additionally, since data is likely to be switched in and out of the partition you will also want a final and empty partition – in which you can repeatedly split – in order to make room for a new partition of data to "switch" in to the table. This last 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 point values are needed. To partition the data into five groups you will define the four boundary values for the partitions and then define whether each 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 defined. This may sound confusing but a partition function includes all data values (-∞ on the far left and ∞ to the far right). By specifying LEFT or RIGHT you define whether or not the boundary condition is an upper boundary in the first partition (LEFT) or a lower boundary in second partition (RIGHT). In other words if the first value (or boundary condition) of a partition function is '20001001' then the values within the bordering partitions will 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 datetime you always have 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 this type of data then you 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 beginning values (of the second partition set) with RIGHT and ending values (of the first partition set) with LEFT. The three following clauses create logically 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 correct boundary cases. Notice the simplicity with RIGHT – this is easier as the default time is 12:00:00.000am. For LEFT the added complexity is due to the precision of the datetime datatype. The reason that 23:59:59.997 MUST be chosen is that datetime data does not guarantee precision to the millisecond. Instead, datetime data is precise within 3.33 milliseconds. In the case of 23:59:59.999, this exact time tick is not available and instead the value is rounded to the nearest time tick that is 12:00:00.000am of the following day. With this rounding, the boundaries will not be defined properly. For datetime data, you must use caution with specifically supplied millisecond values.

Note: Partitioning functions also allow functions as part of the partition function definition. You may use DATEADD(ms,-3,'20010101') instead of explicitly defining the time using '20001231 23:59:59.997'.

For more information, refer to the Books Online topic: "Date and Time" in the Transact-SQL Reference.

To store one fourth of the Orders data in the four active partitions (each will represent one calendar quarter) and create a fifth partition for later use (again, as a placeholder for rolling data in and out of the partitioned table), use a LEFT partition function with four boundary 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 the partition function is defined as LEFT and one empty partition to the extreme LEFT when the partition function is defined as RIGHT. Review the data sets created 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 LEFT partition function, the last value will define the last absolute partition value but there must be a place for all values – meaning that a final partition must exist 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 specific filegroups. When you define a partition scheme, you must make sure to name a filegroup for EVERY partition, even if multiple partitions will reside on the same filegroup. For the range partition created previously (OrderDateRangePFN) there are 5 partitions, the last – and empty partition – will be created in the PRIMARY filegroup. There is no need for a special location for this partition because it will never contain 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 to take advantage of them. The table defines which "scheme" should be used and the scheme defines the function. To tie all three of these together you must specify the column on which the partitioning function should apply. Range partitions always map to exactly one column of the table and it should match the datatype of the boundary conditions defined within the partition function. Additionally, if the table should specifically restrict the data set to a more limited set (rather than from –infinity to positive infinity) then a check constraint should be added as well.

```
CREATE TABLE [dbo].[OrdersRange]
(
      [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] 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. When this is true, the index is aligned with the table. It is not required, but often desired to have the table and its indexes aligned. Aligning a table and its indexes allows for easier management and administration especially when working with the sliding window scenario. However, indexes do not need 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 that only the appropriate partition must be verified to guarantee uniqueness. Therefore if you need to partition a table on one column and you have to create unique index on a different column then they cannot be aligned; the index may be partitioned on the unique column (if this is multi-column unique key then this may be any of the key columns) or not partitioned at all. Be aware that this index will need to be dropped and 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 into a non-partitioned, un-indexed table and then create the indexes to partition the data after the load. By defining a clustered index on a partition scheme, you will effectively partition a table after the fact. This is also a great way of partitioning an existing table. To create this same table as a non-partitioned table and create the clustered index as a partitioned clustered index, replace the ON clause in the create 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 process understood. However, for each step, there are specific settings and options that are available and in certain cases, a variety of criteria must be met. This 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 as analysis data. Analysis is often done for monthly, quarterly and/or yearly ranges of data. Because different analysts may want to see large amounts of different data at the same time – partitioning will better isolate their activity. In this scenario, the active data comes from 283 branch locations and is delivered in two standard format ASCII files. All files are placed on a centrally located fileserver no later than 3am on the first day of each month. Each file ranges in size but averages roughly 86,000 sales (Orders) per month. Each order averages 2.63 line items therefore the OrderDetails files averages 226,180 rows. Each month roughly 25 million new Orders and 64 million OrderDetails 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 just under 600 million Orders and just over 1.5 billion OrderDetails rows. Because analysis is often done comparing months within the same quarter or 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 using range partitioning based on OrderDate. Looking at the requirements for this new server, analysts tend to aggregate and analyze up to six consecutive months of data or up to 3 months of the current and past year (for example, January through March 2003 with January through March 2004). To maximize disk striping as well as isolate most groupings of data, multiple filegroups will use the same physical disks but the filegroups will be offset by six months to reduce disk contention. The current data is October 2004 and all 283 stores are managing their current sales locally. The server holds data from October 2002 through the end of September 2004. To take advantage of the new 16-way multiprocessor machine and SAN each month will have its own file in a filegroup and will reside on a striped mirrors (RAID 1+0) set of disks. As for the physical layout of data to logical drive – via filegroups, the following diagram (Figure 12) depicts where the data resides based on month.

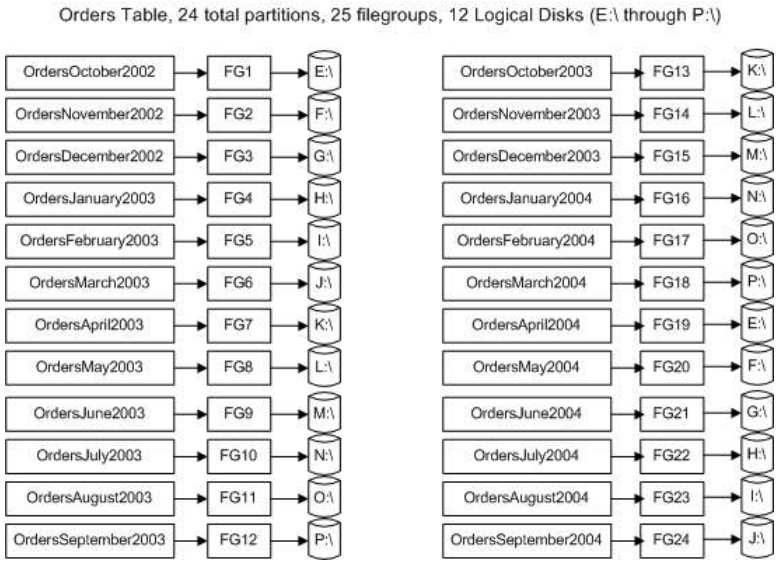

Figure 12: Partitioned Table Orders

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

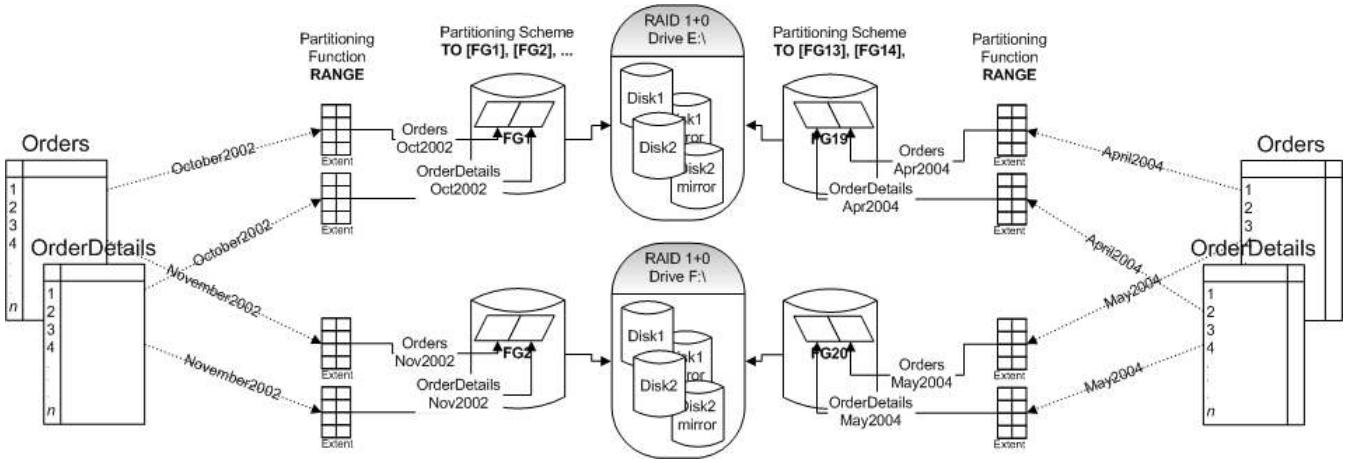

Figure 13: Range Partition to Extent Placement on Disk Arrays

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

```
ALTER DATABASE SalesDB
ADD FILE
      (NAME = N'SalesDBFG1File1',
        FILENAME = N'E:\SalesDB\SalesDBFG1File1.ndf',
        SIZE = 20GB,
        MAXSIZE = 35GB,
  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 partition scheme. To 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 need to store the OrderDate in order to partition both tables on this value. In effect, the partitioning key value (if both expect to be partitioned on the same key value) will be duplicated information; however, it is necessary for the alignment benefits and (in most cases) should be a relatively narrow column (the datetime date type is 8 bytes). As described in *CREATE PARTITION FUNCTION for a Range Partition* the function will be 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 table will maintain a 25th partition that will remain empty. No special filegroup is needed for this empty partition, as no data should ever reside in it. To ensure that no data resides within this 25th filegroup a constraint will restrict the table's data. To direct the data to the appropriate disks

a partition scheme is used to map the partition to the filegroup. The partition scheme will use an explicit filegroup name 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
( [FG1], [FG2], [FG3], [FG4], [FG5], [FG6],
                          [FG7], [FG8], [FG9], [FG10],[FG11],[FG12],
                          [FG13],[FG14],[FG15],[FG16],[FG17],[FG18],
                          [FG19],[FG20],[FG21],[FG22],[FG23],[FG24],
                          [PRIMARY] )
GO
```

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

```
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 [OrderDetailsModifiedDateDFLT]
                  DEFAULT (getdate()),
      [LineTotal]  AS (([UnitPrice]*[OrderQty])),
      [StockedQty]  AS (([ReceivedQty]-[RejectedQty]))
) ON TwoYearDateRangePScheme(OrderDate)
GO
```

The next step to load the data is simply handled through two INSERT statements. These two insert statements use the new AdventureWorks database from which to copy data. Make sure to install the AdventureWorks sample database in order to copy this data:

```
INSERT dbo.[Orders]
      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 the partition on which the data will reside. The following query is helpful as it returns the following information about each partition that contains data: how many rows exist within each partition, the minimum and maximum OrderDate. A partition that does not contain rows will not be 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]
GO

SELECT $partition.TwoYearDateRangePFN(od.OrderDate)
                  AS [Partition Number]
      , min(od.OrderDate) AS [Min Order Date]
      , 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 OrderDetails and Orders. In this case, the clustered index will be defined on the Primary Key as you will also identify both of these tables by their partitioning key (and for OrderDetails you will add the LineNumber to the index for uniqueness). When created, the default behavior of indexes built on partitioned tables is to align the index with the partitioned table on the same scheme; the scheme does not need to be specified.

```
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 where the individual partitions are joined first and then the subsets are added together. Moreover, regardless of how the partitions are joined, SQL Server always 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 going to return information from the third quarter only. Typically, the third quarter includes slower months for order processing; however, in 2004 these months were some of the largest months for Orders. In this case, we want to see if there are any interesting trends related to Products, 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's Primary Key is a composite key of both OrderDate and OrderID the join between these tables will show that OrderDate must be equal between the tables. 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 dbo.Orders AS o
      INNER JOIN dbo.OrderDetails AS od
            ON o.OrderID = od.OrderID AND o.OrderDate = od.OrderDate
WHERE o.OrderDate >= '20040701'
            AND o.OrderDate <= '20040930 11:59:59.997'
GO
```

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

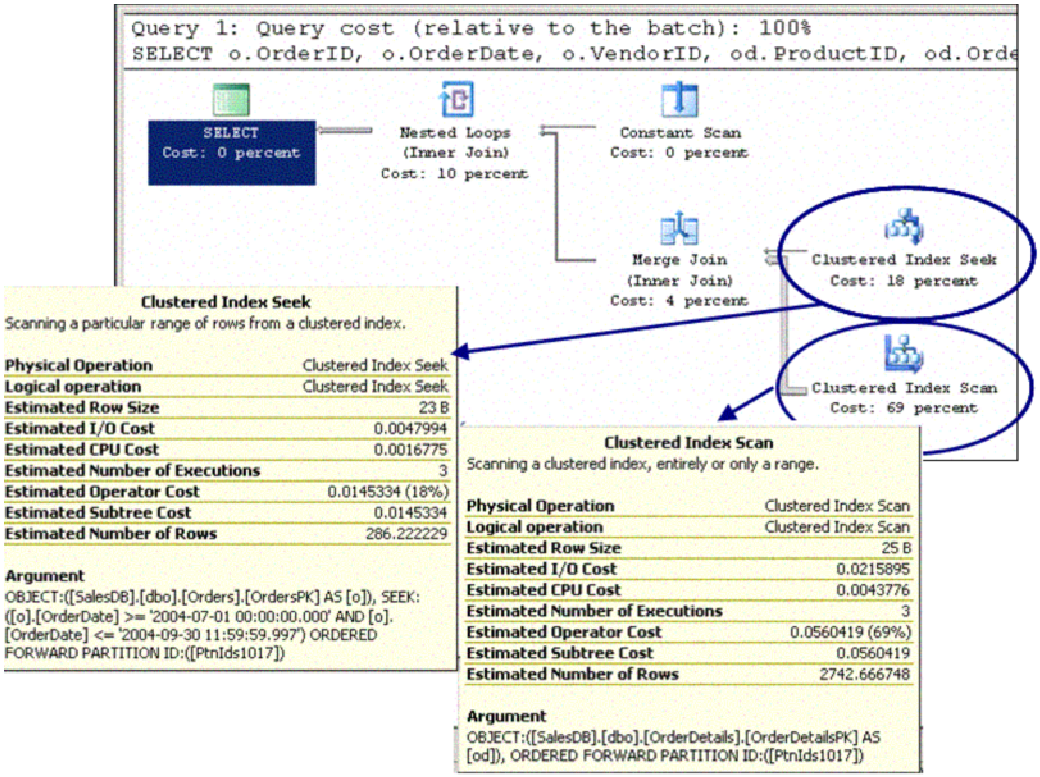

Figure 14: Number of Executes

Shown in figure 15, SQL Server is eliminating all unnecessary partitions and choosing only those that contain the correct data review the **PARTITION ID:([PtnIds1017])** within the Argument section. You may wonder from where the PtnIds1017 expression comes. 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)))**. This represents the partition numbers.

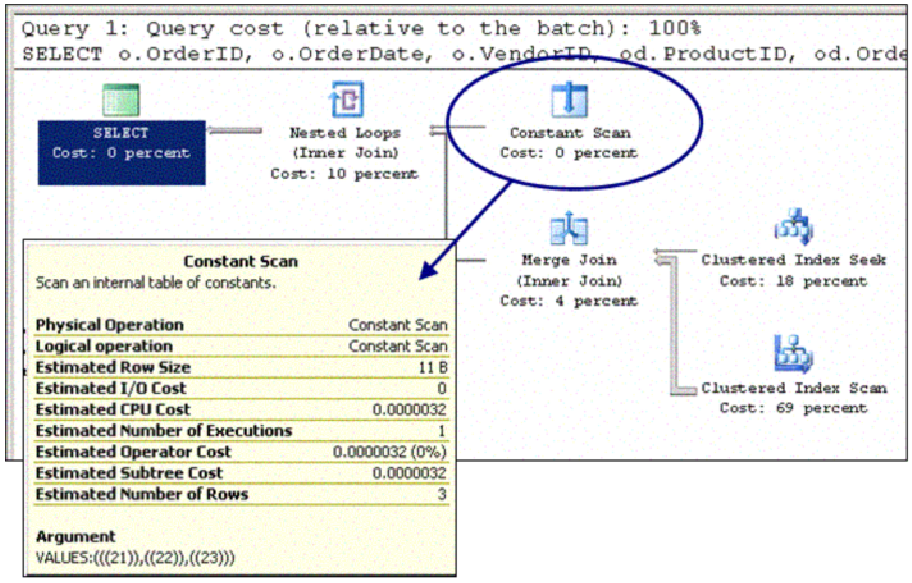

Figure 15: Number of Executes

To verify what data exists on those partitions and only those partitions you can use a slightly modified version of the query we used earlier 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 can be used for partitioned tables and indexes – specifically if they are aligned with tables to which you are joining. SQL Server may perform "multiple" joins by joining 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 remaining partitions – individually. In addition to seeing the number of executes for each table access you should also notice the information related to the "merge" join. If you hover over the merge join, you can see that the merge join was executed three times as well.

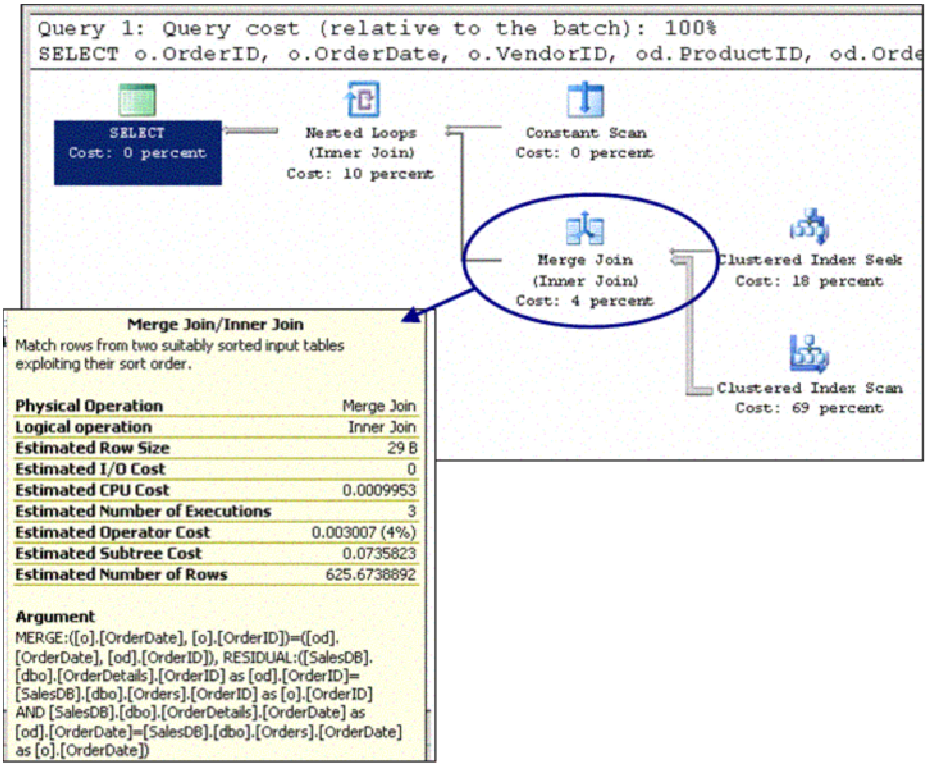

Figure 16: Joining Partitioned Tables

In figure 16, notice there's an additional "nested loops" join performed. It appears as though this happens after the merge join but it turns out that this join is superfluous. The partition ids were already passed into each table seek/scan and this final join only brings together the two portioned sets of data making sure that each adheres to the partition id defined at the start (in the constant scan expression).

Sliding Window Scenario

Once the next month's data is available (in this case, October 2004) there is a specific sequence of steps you must follow in order to use the existing filegroups and "slide" or "switch" the data in and out. In this sales scenario, the data currently in FG1 is data from October 2002. Now that October 2004's data is available, you have two options depending on available space and archiving requirements. Remember, in order to switch a partition in or out of a table quickly, the switch must change metadata ONLY. In other words, the new table (the source or destination – i.e., a partition in disguise) must be created on the same filegroup being switched in or out. If you plan to continue to use the same filegroups, FG1 in this case, then you will need to determine how to handle the space and archiving requirements. If you want to only have a minimal amount of time where the table does not have a complete two years worth of data AND you have the space – you can load the current data (October 2004) into FG1 without removing the data to be archived (October 2002). However, if you do not have enough 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 backup the filegroup immediately after the new partition is loaded and switched in (not just before you plan to switch it out). The reason this is best is if there were a failure on the RAID array, the filegroup could be restored rather than having to rebuild/reload the data. In this case specifically, since the database was only recently partitioned – you could have performed a full backup after the partitioning structure was stabilized (a full database backup is not your only option however, there are numerous backup strategies which can be implemented in SQL Server 2005 and many offer better granularities for backup and restore). For future backups because so much of the data is **not** changing, you can backup the individual filegroups after they are loaded. In fact, this should become part of your rolling partition strategy. For more information, refer 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 of steps may seem complex however each month the overall process will be identical. All that will change are the dates. By using dynamic SQL execution, 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 notes to aid in 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 have a constraint to restrict its data to only valid data for the soon-to-be partition of the table. However, for better performance you will want to load the data into an unindexed heap with no constraints and then add the constraint (see step 3) WITH CHECK before switching the table into 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. Depending on when your process runs detecting month is easy with built-in functions such as DATENAME(m, getdate()). The table's structure must match the existing table – so the primary change for each month is to the table name. However, you could use the same name each month as this table does not need to exist after it is added to the partition. It does still exist after you switch the data into the partitioned table but you can drop the staging table once the switch has been completed. Additionally, the date range must change. Since you are working with datetime data and there are rounding issues with regard to how time is stored, you must be able to determine programmatically the proper millisecond value. The easiest way to find the last datetime value for the end of the month is to take the month with which you are working, add 1 month to it and then subtract 2 or 3 milliseconds from it. You cannot subtract only 1 millisecond as 59.999 rounds up to .000 - which is really the first day of the next month. You can subtract 2 or 3 milliseconds, as -2 milliseconds rounds down to .997 and 3 milliseconds equals .997; .997 is a valid value that can be stored. This will give you the correct ending value for your datetime range:

```
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 switching in and out. To determine the appropriate filegroup with which to work, use the following system table query combined with the **$partition** function shown earlier. Specify any date within the range being rolled out. This is the partition (and filegroup) in which all of the work will be performed. The underlined sections will need to be changed for your specific table, partitioning function and specific date.

```
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 statements.
   In automation: this process is the most complex to automate. You will need to make sure that all files have been loaded and you may want to load them in parallel. To control this process you might consider a table that keeps track of which files have been loaded and where the files are located. You can create a SQL Agent job that checks for files every few minutes, picks up the new files and then 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 CHECK. The WITH CHECK setting is the default so it does not need to be specified but it is important NOT to say WITH NOCHECK.

```
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
```

   In automation: this is an easy step. Using the month and filegroup information from step 1, you can create this clustered index.

Manage a second Staging Table for the partition that will be switched OUT

1. Create a second staging table – this is an empty table that will hold the partition's data when it is switched out.

```
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 (and the partition will 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 a part of the partition scheme. Since you will roll the new data through the same existing 24 partitions then you will need to make FG1 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()
SPLIT RANGE ('20041031 23:59:59.997')
GO
```

5. Change the constraint definition on the base table (if one exists) to allow for the new range of data. Since adding constraints can be costly (to verify the data), there is a best practice in how you do this, which can make it more efficient. The best practice is to keep extending the date instead of dropping and re-creating the constraint. For now, only one constraint exists (OrdersRangeYearCK) but 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 fastest way to remove 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 has been chosen then a file or filegroup backup should be performed. If a full database based backup strategy has been chosen then a full database 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 periodically, consider using defined "range" partitions in the form of a list. In other words, you will use a function that explicitly defines each partition as a value for a region. For example, consider a Spanish company that has clients in Spain, France, Germany, Italy, and the UK. Their sales 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 boundaries do not have any other values outside of the actual partition key. Instead, it is a list – not ranges. Although a list, the boundary conditions must include the extreme left and the extreme right. To create 5 partitions you should specify only 4 in the partition function. The values do not need to be ordered but the easiest way to get the correct number of partitions is to order the partition values and leave off the last partition (for LEFT) or to order the partition values and start with the second value (for RIGHT).

Because there are five partitions, you must have five filegroups. In this case, the filegroups will be named after the data being stored. The script file RegionalRangeCaseStudyFilegroups.sql is a script that shows all of this syntax. Each filegroup is created using the same settings, yet they do not have to be if the data is not balanced. Only the filegroup and file for Spain are shown; each of the four additional filegroups and files have the same parameters yet exist on different drives and have the specific name of the country partition.

```
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. In this case, the list 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. All five filegroups 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 partitions, for example – list partitions.

Summary

SQL Server 2005 offers a way to manage - easily and consistently - large tables and indexes through partitioning. By partitioning large tables and indexes you have a way to manage subsets of your data outside of the partition – allowing simplified management, increased performance and abstracted application logic; the partitioning scheme is entirely transparent to the application. When your data has logical groupings (ranges or lists) and larger queries must analyze that data within these predefined and consistent ranges as well as manage incoming and outgoing data within the same predefined ranges, then range partitioning is a simple choice. If you are looking at analysis 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 multiple filegroups 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. Each file within the zip is described below:

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

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

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

RangeCaseStudyScript4-SlidingWindow.sql – Includes the syntax and process associated with the monthly management of the range partitioned table case study. In this script, you will "slide" data both in and out of the Orders table. Optionally, on your own – create the same process to move data in and out for the OrderDetails table. Hint: see the Insert used in RangeCaseStudyScript2 for the table and correct columns of data to insert for OrderDetails.

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