Microsoft SQL Server 9.0 Technical Articles

Partitioned Tables and Indexes in SQL Server 2005

Kimberly L. Tripp
Founder, SQLskills.com

January 2005

Applies to:
SQL Server 2005

Summary: Table-based partitioning features in SQL Server 2005 provide flexibility and performance to simplify the creation and maintenance of partitioned tables. Trace the progression of capabilities from logically and manually partitioning tables to the latest partitioning features, and find out why, when, and how to design, implement, and maintain partitioned tables using SQL Server 2005. (41 printed pages)

About this paper The features and plans described in this document are the current direction for the next version of the SQL Server. They are not specifications for this product and are subject to change. There are no guarantees, implied or otherwise, that these features will be included in the final product release.

For some features, this document assumes that the reader is familiar with SQL Server 2000 features and services. For background information, see the SQL Server Web site [http://www.microsoft.com/sql/] or the SQL Server 2000 Resource Kit [http://www.microsoft.com/mspress/books/4939.asp] .

This is not a product specification.

Download the associated code sample, SQL2005PartitioningScripts.exe [http://download.microsoft.com/download/a/b/4/ab490b9f-7ad5-45ef-9f03-428e24076776/sql2005partitioningscripts.exe] .

Contents

Why Partition?

What are partitions and why use them? The simple answer is: To improve the scalability and manageability of large tables and tables that have varying access patterns. Typically, you create tables to store information about an entity, such as customers or sales, and each table has attributes that describe only that entity. While a single table for each entity is the easiest to design and understand, these tables are not necessarily optimized for performance, scalability, and manageability, particularly as the table grows larger.

What constitutes a large table? While the size of a very large database (VLDB) is measured in hundreds of gigabytes, or even terabytes, the term does not necessarily indicate the size of individual tables within the database. A large database is one that does not perform as desired or one in which the operational or maintenance costs have gone beyond predefined maintenance or budgetary requirements. These requirements also apply to tables; a table can be considered large if the activities of other users or maintenance operations have a limiting affect on availability. For example, the sales table is considered large if performance is severely degraded or if the table is inaccessible during maintenance for two hours each day, each week, or even each month. In some cases, periodic downtime is acceptable, yet it can often be avoided or minimized by better design and partitioning implementations. While the term VLDB applies only to a database, for partitioning, it is more important to look at table size.

In addition to size, a table with varying access patterns might be a concern for performance and availability when different sets of rows within the table have different usage patterns. Although usage patterns may not always vary (and this is not a requirement for partitioning), when usage patterns do vary, partitioning can result in additional gains in management, performance, and availability. Again, to use the example of a sales table, the current month's data might be read-write, while the previous month's data (and often the larger part of the table) is read-only. In a case like this, where data usage varies, or in cases where the maintenance overhead is overwhelming as data moves in and out of the table, the table's ability to respond to user requests might be impacted. This, in turn, limits both the availability and the scalability of the server.

Additionally, when large sets of data are being used in different ways, frequent maintenance operations are performed on static data. This can have costly effects, such as performance problems, blocking problems, backups (space, time, and operational costs) as well as a negative impact on the overall scalability of the server.

How can partitioning help? When tables and indexes become very large, partitioning can help by partitioning the data into smaller, more manageable sections. This paper focuses on horizontal partitioning, in which large groups of rows will be stored in multiple separate partitions. The definition of the partitioned set is customized, defined, and managed by your needs. Microsoft SQL Server 2005 allows you to partition your tables based on specific data usage patterns using defined ranges or lists. SQL Server 2005 also offers numerous options for the long-term management of partitioned tables and indexes by the addition of features designed around the new table and index structure.

Furthermore, if a large table exists on a system with multiple CPUs, partitioning the table can lead to better performance through parallel operations. The performance of large-scale operations across extremely large data sets (for instance many million rows) can benefit by performing multiple operations against individual subsets in parallel. An 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, queries joining large datasets 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 (such as **Order** and **OrderDetails** tables) 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 join the data that resides on the same partitions first and then combine the results. This allows SQL Server 2005 to more effectively use multiple-CPU computers.

History of Partitioning

The concept of partitioning is not new to SQL Server. In fact, forms of partitioning have been possible in every release of the product. However, without features specifically designed to help you create and maintain a partitioning scheme, partitioning has often been cumbersome and underutilized. Additionally, users and developers misunderstand the schema (due to a more complex database design) and the benefits are diminished. However, because of the significant performance gains inherent in the concept, SQL Server 7.0 began improving the feature by enabling forms of partitioning through partitioned views. SQL Server 2005 now offers the greatest advances for partitioning large datasets through partitioned tables.

Partitioning Objects in Releases Before SQL Server 7.0

In SQL Server 6.5 and earlier, partitioning had to be part of your design and 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 design complexity. 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 and application access, the query processor had to access every underlying table in order to determine what data was needed for the result set. If only a limited subset of the 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 the 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.

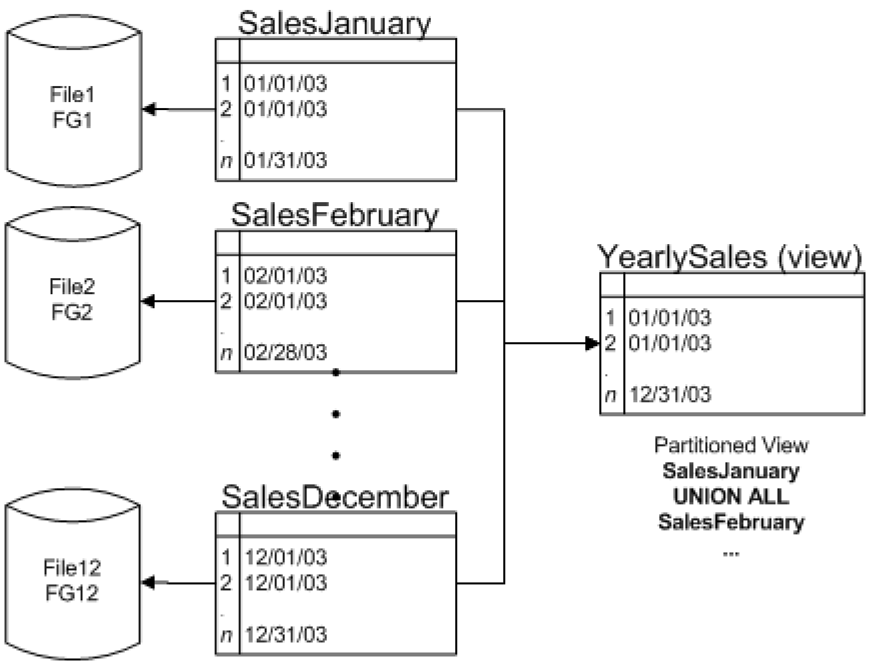


Figure 1. Partitioned views in SQL Server 7.0/2000

Users who access the YearlySales view with the following query will be directed only to the **SalesJanuary2003** table.

Copy Code

```
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. By using this procedure to create trusted constraints, developers can significantly reduce the complexity of their design using views without having to directly access (or even be aware of) the table in which they were interested. With trusted constraints, SQL Server improves performance by removing unnecessary tables from the execution plan.

Note A constraint can become "untrusted" in various ways; for instance, if a bulk insert is performed without specifying the CHECK_CONSTRAINTS argument or if a constraint is created with NOCHECK. If a constraint is 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. In SQL Server 2000, data modification statements also benefit from the partitioned view capabilities introduced in SQL Server 7.0. Because data modification statements can use the same partitioned view structure, SQL Server can direct modifications to the appropriate base table through the view. Additional restrictions on the partitioning key and its creation are required in order to configure this properly; 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, see Using Partitions in a Microsoft SQL Server 2000 Data Warehouse [http://msdn2.microsoft.com/en-us/library/aa902650.aspx] .

Partitioned Tables in SQL Server 2005

While the improvements in SQL Server 7.0 and SQL Server 2000 significantly enhanced performance when using partitioned views, they did not simplify the administration, design, or development of a partitioned data set. When 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 partitioning strategies in previous versions 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.

In SQL Server 2005, each partition has the same indexes by definition. For example, consider a scenario in which the current month of Online Transaction Processing (OLTP) data needs to be moved into an analysis table at the end of each month. The analysis table (to be used for read-only queries) is a single table with one clustered index and two nonclustered indexes; the bulk load of 1 gigabyte (GB) (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, and sacrifices concurrency for speed.

If this data were isolated into a newly created (empty) and unindexed [heap] table, the load could occur first and then the indexes could be created after loading the data. Often you will realize gains in performance of ten times or better by using this scheme. In fact, by loading into an unindexed table you can take advantage of multiple CPUs by loading multiple data files in parallel or by loading multiple chunks from the same file (defined by starting and ending row position). Since both operations can benefit from parallelism, this can yield even further gains in performance.

In any release of SQL Server, partitioning allows you this more granular control, and does not require that you have all data in one location; however, there are many objects to create and manage. 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 process takes only a short period of time and can be made more efficient by 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 queried table until the partition is added. The result is that, typically, adding a partition takes only a few seconds.

The performance improvement is also significant when data needs to be removed. If one database needs a sliding-window set of data in which new data is migrated in (for example, the current month), and the oldest data is removed (maybe the parallel month from the previous year), the performance of this data migration can be improved by orders of magnitude through the use of partitioning. While this may seem extreme, consider the difference without partitioning; when all of the data is in a single table, deleting 1 GB of old 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, does not allow log truncation for the duration of the delete (note that the delete is a single auto-commit transaction; however, you can control the size of the transaction by performing multiple deletes where possible) and therefore requires a potentially much larger log. Using partitioning, however, removing that same data requires removing the specific partition from a partitioned table (which is a metadata operation) and then dropping or truncating 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 physical location of data cannot be predicted. For systems where parallelism is not expected, SQL Server improves performance by using all disks more evenly through filegroups, making specific placement of data less critical.

> **Note** 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 allocations (64-KB chunks, which equals eight 8-KB Pages) 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 File 1 and then another extent to the **OrderDetails** table from File 2. It is likely that the **OrderDetails** table will grow more quickly than the **Orders** table, and the allocations that follow will go to the next table that needs space. As **OrderDetails** grows, it will get the next extent from File 3 and SQL Server continues to "round robin" through the files within the filegroup. In Figure 2, 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.

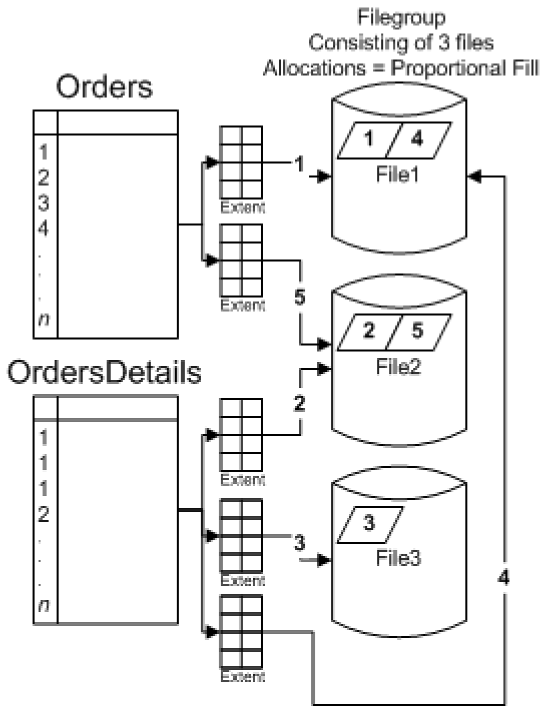


Figure 2. Proportional fill using filegroups

SQL Server continues to balance allocations among all of the objects within that filegroup. While SQL Server runs more effectively when using more disks for a given operation, using more disks is not as optimal from a management or maintenance perspective, particularly when usage patterns are very predictable (and isolated). Since the data does not have a specific location on disk, you don't have the ability to isolate the data for maintenance such as backup operations.

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, as well as the partition in which 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 (partition functions will be covered in more detail later in 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 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 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 could 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 other similar construct where equi-join predicates may occur. Collocation is valuable because queries that join tables on the partition columns are generally much faster. Consider the **Orders** and **OrderDetails** tables described in Figure 2. 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 define them to use the same scheme. Related data that has the same value for the partition key will be placed within the same file, 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 storage-aligned because the same data resides within the same files.

Figure 3 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. For example, all of the sales data for January (for both the **Orders** and **OrderDetails** tables) is on the first filegroup, February data is on the second filegroup, and so forth.

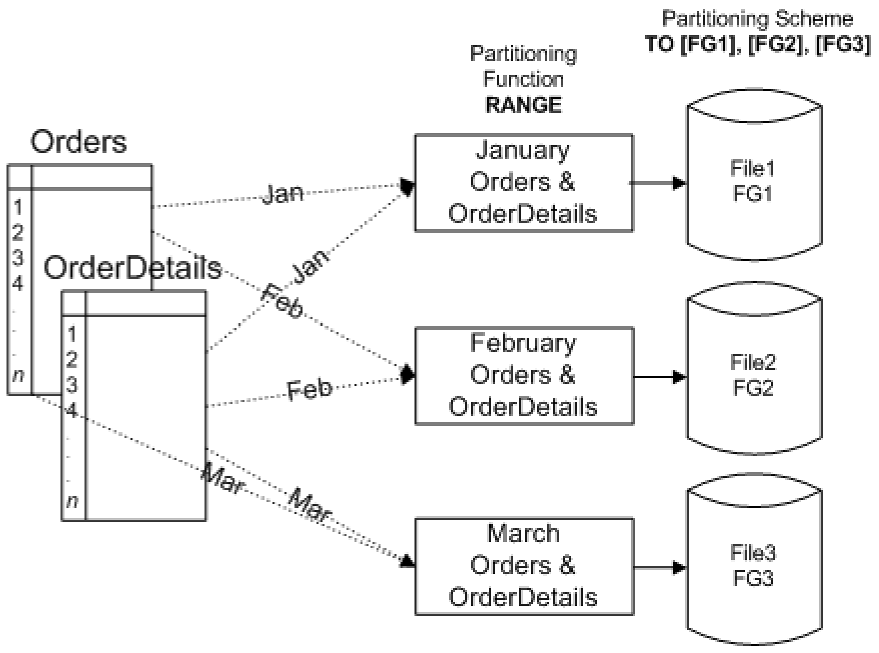


Figure 3. Storage-aligned tables

SQL Server allows partitioning based on ranges, and tables and indexes can be designed to 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 the design and implementation of large tables that need to be partitioned for performance or manageability purposes.
- Load data into a new partition of an existing partitioned table with minimal disruption in data access in the remaining partitions.
- Load data into a new partition of an 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. To understand these new concepts, let's first review a table's structure with regard to creation and placement. 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 be created on a partitioning scheme. The partitioning scheme maps the object to one or more 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 that are defined by specific and customizable ranges of data. The range partition boundaries are chosen by the developer, and can be changed as data usage patterns change. Typically, these ranges are date-based or based on 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, such as a given month or quarter), and for combined OLTP and Decision Support Systems (DSS) where data usage varies over the life cycle of a row. The biggest benefit to a SQL Server 2005 partitioned table and index 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 replaced very quickly. Range partitions are best suited when data access is typically for decision support over large ranges of data. In this case, it matters where the data is specifically located so that only the appropriate partitions are accessed, when necessary. Additionally, as transactional data becomes available you can add the data 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 one or more filegroups. However, they often follow a consistent pattern so once defined, they will likely be easy to maintain programmatically (see Figure 4).

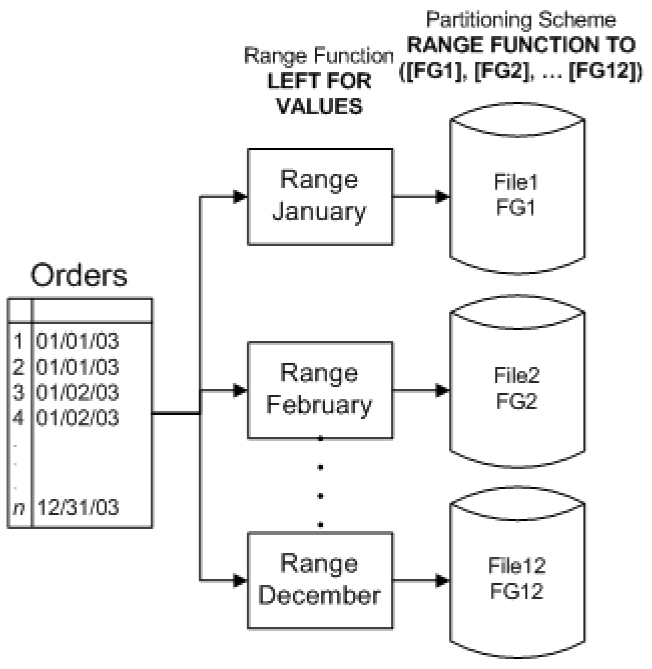


Figure 4. Range partitioned table with 12 partitions

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 single column in the table and must meet certain criteria. The partition function defines the data type on which the key (also known as the logical separation of data) is based. The function defines this key but not the physical placement of the data on disk. The placement of data is determined by the partition scheme. In other words, the scheme maps the data to one or more filegroups that map 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 dataset is divided by a logical and data-driven boundary. In fact, the data partitions may not be truly balanced. Data usage dictates a range partition when the table is used in a pattern that defines specific boundaries of analysis (also known as 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 exist within the table (due to data integrity/constraints). 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). Therefore, to restrict the domain of values to a specific dataset, partitions must be combined with CHECK constraints. Using check constraints to enforce your business rules and data integrity constraints allows you to restrict the dataset to a finite range rather than infinite range. Range partitions are ideal when maintenance and administration requires archiving large ranges of data on a regular basis and when queries access a large amount of data that is within a subset of the ranges.

Index Partitioning

In addition to partitioning a table's dataset, you can partition indexes. Partitioning both the table and its indexes using the same function often optimizes performance. When the indexes and the table use the same partitioning function and columns in the same order, the table and index are 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, because all 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, they are considered to be *storage-aligned*. One benefit to storage alignment is that all data within the same boundary is located on the same physical disk(s). In this case, backups can be isolated to a certain timeframe and your strategies can vary, in terms of frequency and backup type, based on the volatility of the data. Additional gains can be seen when tables and indexes on the same file or filegroup are joined or aggregated. SQL Server benefits from parallelizing an operation across partitions. In the case of storage alignment and multiple CPUs, each processor can work directly on a specific file or filegroup with no conflicts in data access because all required data is on the same disk. This allows more processes to run in parallel without interruption.

For more details, see "Special Guidelines for Partitioned Indexes" in SQL Server Books Online [http://www.microsoft.com/sql/techinfo/productdoc/2000/books.asp] .

Special Conditions for Partitions: Split, Merge, and Switch

To aid in the usage of partitioned tables are several new features and concepts related to partition management. Because partitions are used for large tables that scale, the number of partitions chosen when the partition function was created changes over time. You can use the ALTER TABLE statement with the new split option to add another partition to the table. When a partition is split, data can be moved to the new partition; but to preserve performance, rows should not move. This scenario is described in the case study later in this paper.

Conversely, to remove a partition, perform a switch-out for the data and then merge the boundary point. 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) when the data for the current month becomes available. For example, you might choose to have one year of data available, and at the end each month you switch in the current month and then switch out the earliest month, differentiating between the current month's read/write OLTP versus the previous month's read-only data. There is a specific flow of actions that makes the process most efficient, as illustrated by the following scenario.

You are keeping a year's worth of read-only data available. Currently, the table holds data from September 2003 through 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 that is empty. This last partition is empty because a range partition always includes the entire domain—both the extreme left as well as the extreme right. And if you plan to manage data in a sliding-window scenario, you'll always want to have an empty partition to split into which new data will be placed. In a partition function defined with LEFT boundary points, the empty partition will logically exist to the far RIGHT. Leaving a partition empty at the end will allow you to split the empty partition (for the new data coming in) and not need to move rows from the last partition (because none exist) to the new filegroup that's being added (when the partition is split to include another chunk of data). This is a fairly complex concept that will be discussed in greater detail in the case study later in the paper but the idea is that all data additions or deletions should be metadata-only operations. To ensure that metadata-only operations occur, you will want to strategically manage the changing part of the table. To ensure that this partition is empty, you will use a check constraint to restrict this data in the base table. In this case, the **OrderDate** should be on or after September 1, 2003 and before September 1, 2004. If the last defined boundary point is August 31 at 11:59:59.997 (more on why 997 is coming up), then the combination of the partition function and this constraint will keep the last partition empty. 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.

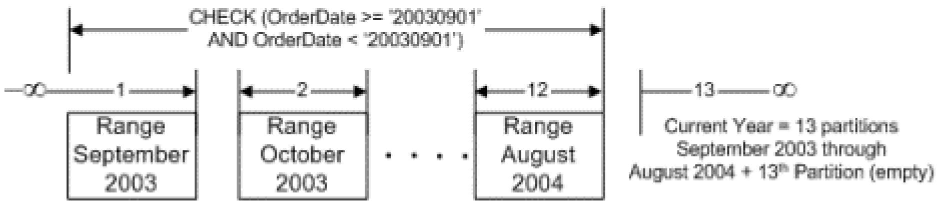

CHECK (OrderDate >= '20030901'
AND OrderDate < '20030901')
-∞
1
2
12
13
∞
Range September 2003
Range October 2003
Range August 2004
Current Year = 13 partitions
September 2003 through
August 2004 + 13th Partition (empty)

Figure 5. Range partition boundaries before data load/archive

When October begins (in the OLTP database), September's data should move to the partitioned table, which is 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 be using "staging tables" that will eventually become partitions within the partitioned table A detailed description of this scenario is explained later in the case study later in this paper. In this process, you will switch out (Figure 6) a partition of a table to a nonpartitioned table within the same filegroup. Because the nonpartitioned table already exists within the same filegroup (and this is critical for success), SQL Server can make this switch as a metadata change. As a metadata-only change, this can occur within seconds 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.

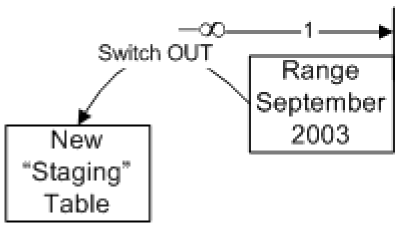


Figure 6. Switching a partition out

To remove the oldest partition (September 2003), use the new merge option (shown in Figure 7) with ALTER TABLE. Merging a boundary point effectively removes a boundary point, and therefore a partition. This reduces the number of partitions into which data loads to *n-1* (in this case, 12). Merging a partition should be a very fast operation when no rows have to be moved (because the boundary point being merged has no data rows). In this case, because the first partition is empty, none of the 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 have to move from the first to the second partition, which can be a very expensive operation. However, this is avoided in the most common sliding-window scenario where an empty partition is merged with an active partition, and no rows move.

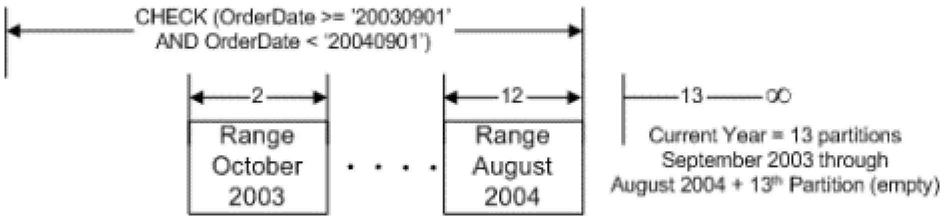

CHECK (OrderDate >= '20030901' AND OrderDate < '20040901')
2
Range October 2003
12
Range August 2004
13——∞
Current Year = 13 partitions September 2003 through August 2004 + 13th Partition (empty)

Figure 7. Merging a partition

Finally, the new table has to be switched in to the partitioned table. In order for this to be performed as a metadata change, loading and building indexes must happen in a new table, outside of the bounds of the partitioned table. To switch in the partition, first split the last, most recent, and empty range into two partitions. Additionally, you need to update the table's constraint to allow the new range. Once again, the partitioned table will have 13 partitions. In the sliding window scenario, the last partition with a LEFT partition function will always remain empty.

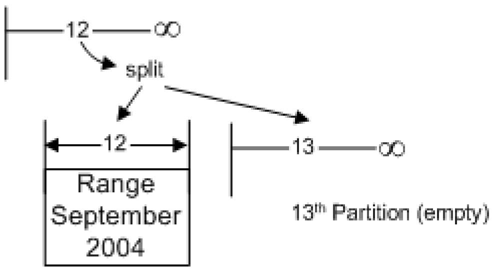


Figure 8. Splitting a partition

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

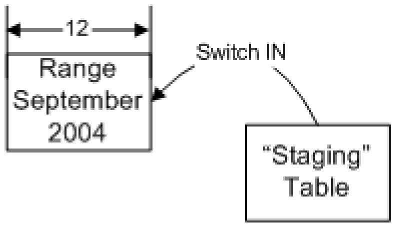


Figure 9. Switching a partition in

The result of the table is:

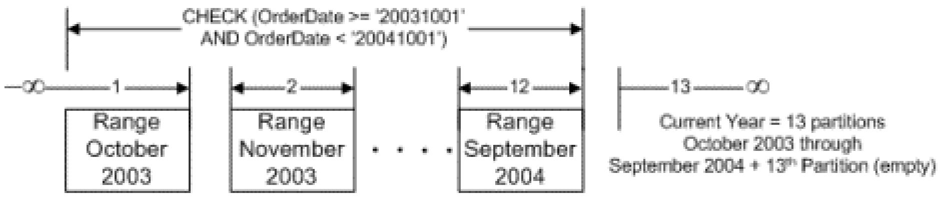

CHECK (OrderDate >= '20031001' AND OrderDate < '20041001')
—∞——1
Range October 2003
2
Range November 2003
12
Range September 2004
13——∞
Current Year = 13 partitions October 2003 through September 2004 + 13th Partition (empty)

Figure 10. Range partition boundaries after data load/archive

Because only one partition can be added or removed at a time, tables that need to have more than one partition added or removed should be recreated. To change to this new partitioning structure, first create the new partitioned table and then load the data into the newly created table. This is a more optimal approach than rebalancing the whole table for each split. This process is accomplished by using a new partition function, a new partition scheme, and then by moving the data to the newly partitioned table. To move the data, first copy the data using INSERT *newtable* SELECT *columnlist* FROM *oldtable* and then drop the original tables. Prevent user modifications while this process is running to help ensure against data loss.

For more details, see "ALTER PARTITION FUNCTION" and "ALTER TABLE" in SQL Server Books Online [http://www.microsoft.com/sql/techinfo/productdoc/2000/books.asp] .

Steps for Creating Partitioned Tables

Now that you have an understanding of the value of partitioned tables, the next section details the process of implementing a partitioned table, and the features that contribute to this process. The logical flow is as follows:

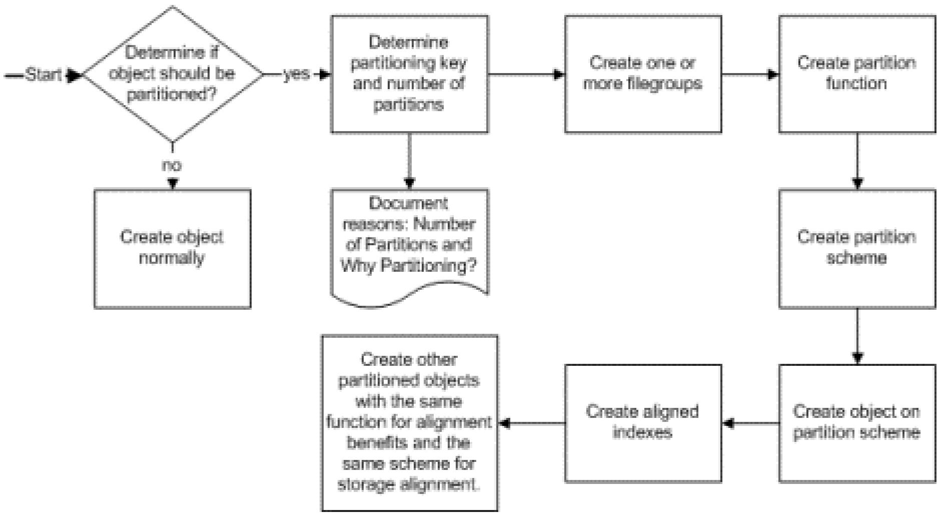

Start
Determine if object should be partitioned?
yes
no
Determine partitioning key and number of partitions
Create one or more filegroups
Create partition function
Create object normally
Document reasons: Number of Partitions and Why Partitioning?
Create partition scheme
Create other partitioned objects with the same function for alignment benefits and the same scheme for storage alignment
Create aligned indexes
Create object on partition scheme

Figure 11. Steps to create a partitioned table or index

Determine If Object Should Be Partitioned

While partitioning can offer great benefits, it adds administrative overhead and complexity to the implementation of your objects, which can be more of a burden than a gain. Specifically, you might not want to partition a small table, or a table that currently meets performance and maintenance requirements. The sales scenario mentioned earlier uses partitioning to relieve the burden of moving rows and data—you should consider whether your scenario has this sort of burden when deciding whether to implement partitioning.

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, range partitioning can alleviate contention as well as reduce maintenance when read-only data does not require it. To determine the number of partitions, you should evaluate whether or not logical groupings and patterns exist within your data. If 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. only the specific partition).

For more details, see "Designing Partitioned Tables and Indexes" in SQL Server Books Online [http://www.microsoft.com/sql/techinfo/productdoc/2000/books.asp] .

Determine If Multiple Filegroups Should Be Used

To help optimize performance and maintenance, you should use filegroups to separate your data. The number of filegroups is partially dictated by hardware resources: it is generally best to have the same number of filegroups as partitions, and these filegroups often reside on different disks. However, this primarily only pertains to systems where analysis tends to be performed over the entire dataset. 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 I/O balancing, you will need to create at least one filegroup. 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 so that only related or logically grouped data resides on the same filegroup. Using ALTER DATABASE, you can add a logical filegroup name, and then add files. To create a filegroup named 2003Q3 for the **AdventureWorks** database, use ALTER DATABASE in the following way:

Copy Code

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

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

Copy Code

```
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) by specifying a filegroup in the ON clause of CREATE TABLE. However, a table cannot be created on multiple filegroups unless it is partitioned. To create a table on a single filegroup, use the ON clause of CREATE TABLE. To create a partitioned table, you must first have a functional mechanism for the partitioning. The criteria on which you partition are logically separated from the table in the form of a partition function. This partition function will exist as a separate definition from the table and this physical separation helps because multiple objects can use the partition function. Therefore, the first step in partitioning a table is to create the partition function.

Create the Partition Function for a Range Partition

Range partitions must be defined with boundary conditions. Moreover, no values, from either end of the range, can be eliminated even if a table is restricted through a CHECK constraint. To allow for periodic switching of data into the table, you'll need a final and empty partition

In a range partition, first define the boundary points: for five partitions, define four boundary point values and specify whether each value is an upper boundary of the first (LEFT) or the lower boundary of the second (RIGHT) partition. Based on the left or right designation, you will always have one partition that is empty, as the partition will not have an explicitly defined boundary point.

Specifically, if the first value (or boundary condition) of a partition function is '20001001' then the values within the bordering partitions will be:

For LEFT

first partition is all data <= '20001001'

second partition is all data > '20001001'

For RIGHT

first partition is all data < '20001001'

second 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** has certain implications: you always have both a date and a time. A date with no defined value for time implies a "0" time of 12:00 A.M. If LEFT is used with this type of data, then the data with a date of Oct 1, 12:00 A.M. will end up in the first partition and the rest of October's data in the second partition. Logically, it is best to use beginning values with RIGHT and ending values with LEFT. These three clauses create logically identical partitioning structures:

Copy Code

```
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

Copy Code

```
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

Copy Code

```
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 set up the correct boundary cases. Notice the simplicity with RIGHT because the default time is 12:00:00.000 A.M. For LEFT, the added complexity is due to the precision of the **datetime** data type. 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 12:00:00.000 A.M. 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, see "Date and Time" in the Transact-SQL Reference in SQL Server Books Online [http://www.microsoft.com/sql/techinfo/productdoc/2000/books.asp] .

To store one-fourth of the **Orders** data in the four active partitions, each representing one calendar quarter, and create a fifth partition for later use (again, as a placeholder for sliding data in and out of the partitioned table), use a LEFT partition function with four boundary conditions:

Copy Code

```
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 five partitions. 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 leftmost 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 fourth 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'.

Create the 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 five 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.

Copy Code

```
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:

Copy Code

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

Create the Partitioned Table

With the partition function (the logical structure) and the partition scheme (the physical structure) 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 together, you must specify the column to which the partitioning function should apply. Range partitions always map to exactly one column of the table that should match the datatype of the boundary conditions defined within the partition function. Additionally, if the table should specifically limit the data set (rather than from –infinity to positive infinity), then a check constraint should be added as well.

Copy Code

```
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. Although not required, aligning a table and its indexes allows for easier management and administration, particularly with the sliding-window scenario.

For example, to create unique indexes, the partitioning column must be one of the key columns; this will ensure verification of the appropriate partition 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. In this case, the index might be partitioned on the unique column (if this is multi-column unique key, then it could be any of the key columns) or it might not be partitioned at all. Be aware that this index has 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 nonpartitioned, unindexed table and then creating the indexes to partition the data after the load. By defining a clustered index on a partition scheme, you will effectively partition the table after the load. This is also a great way of partitioning an existing table. To create the same table as a nonpartitioned table, and create the clustered index as a partitioned clustered index, replace the ON clause in the create table with a single filegroup destination. Then, create the clustered index on the partition scheme after the data is loaded.

Putting it all Together: Case Studies

If you have looked at concepts, benefits, and code samples related to partitioning, you might have a good understanding of the process; however, for each step, there are specific settings and options 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 and the prior month's data is used heavily for analysis. Analysis is often done for monthly, quarterly, and/or yearly ranges of data. Because different analysts may want to see large amounts of varying data at the same time, partitioning better isolates this 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 3:00 A.M. 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, so the **OrderDetails** files average 226,180 rows. Each month, roughly 25 million new **Orders** and 64 million **OrderDetails** rows are added and the historical analysis server maintains two years worth of data active for analysis. Two years 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 used. The boundary for each range is monthly.

Using the steps described in Figure 11, the table is partitioned 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 three 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 Storage Area Network 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.

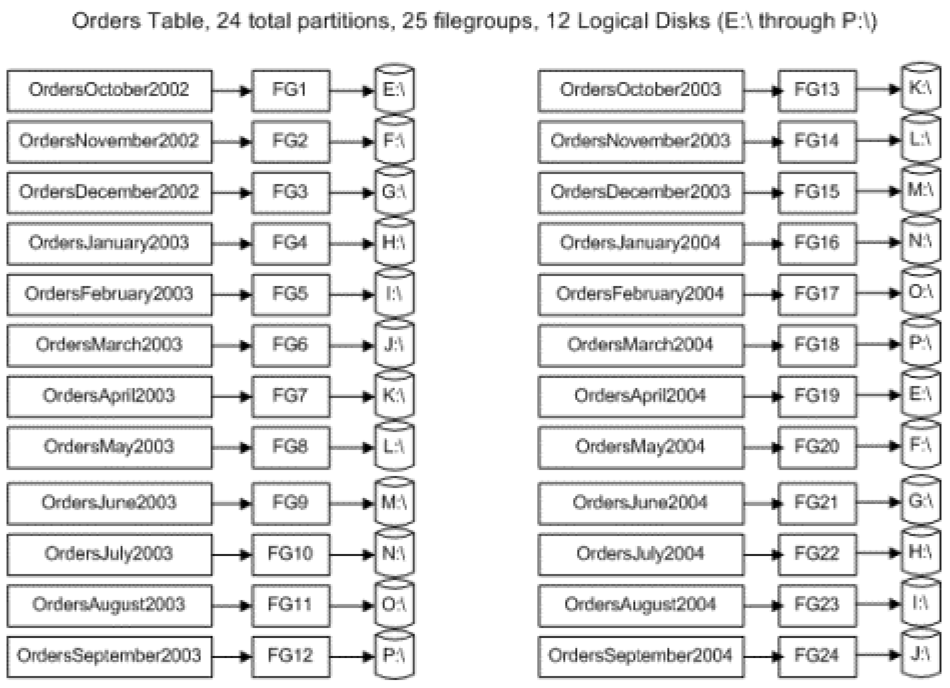
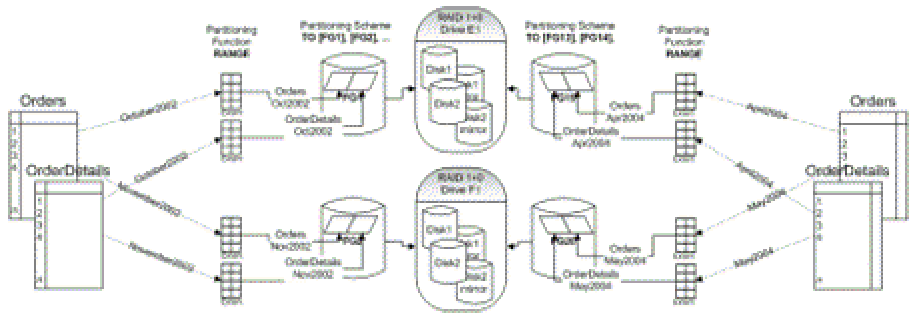

Orders Table, 24 total partitions, 25 filegroups, 12 Logical Disks (E:\ through P:\)
OrdersOctober2002 FG1 E:\
OrdersNovember2002 FG2 F:\
OrdersDecember2002 FG3 G:\
OrdersJanuary2003 FG4 H:\
OrdersFebruary2003 FG5 I:\
OrdersMarch2003 FG6 J:\
OrdersApril2003 FG7 K:\
OrdersMay2003 FG8 L:\
OrdersJune2003 FG9 M:\
OrdersJuly2003 FG10 N:\
OrdersAugust2003 FG11 O:\
OrdersSeptember2003 FG12 P:\
OrdersOctober2003 FG13 K:\
OrdersNovember2003 FG14 L:\
OrdersDecember2003 FG15 M:\
OrdersJanuary2004 FG16 N:\
OrdersFebruary2004 FG17 O:\
OrdersMarch2004 FG18 P:\
OrdersApril2004 FG19 E:\
OrdersMay2004 FG20 F:\
OrdersJune2004 FG21 G:\
OrdersJuly2004 FG22 H:\
OrdersAugust2004 FG23 I:\
OrdersSeptember2004 FG24 J:\

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 the **Orders** and **OrderDetails** data is 48. The Storage Area Network 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, as well as 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:


Partitioning Function RANGE
Partitioning Scheme TO [FG1], [FG2], ...
RAID 1+0 Drive E:\
Partitioning Scheme TO [FG13], [FG14], ...
Partitioning Function RANGE
Orders
OrderDetails
RAID 1+0 Drive F:\

[http://msdn2.microsoft.com/en-us/library/ms345146.sql2k5partition_13(en-us,sql.90).gif]
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. To create both tables as partitioned tables, 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. 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 24 filegroups must be created. See the RangeCaseStudyFilegroups.sql script for a complete script to create all 24 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** database. Each will have the same syntax, with the exception of location, filename, and filegroup name:

Copy Code

```
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 24 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 tables are partitioned on the same key value, will be duplicated information; however, it is necessary for the alignment benefits and, in most cases, it should be a relatively narrow column (the **datetime** date type is 8 bytes). As described in "Create Partition Function for a Range Partition" earlier in this paper, the function will be a range partition function where the first boundary condition will be the in the LEFT (first) partition.

Copy Code

```
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 creates 25 partitions. The table will maintain a twenty-fifth partition that will remain empty. No special filegroup is needed for this empty partition because no data will ever reside in it, as a constraint restricts 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 twenty-fifth and empty partition.

Copy Code

```
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 by using the default filegroup or a user-defined filegroup as a nonpartitioned table, or by using a scheme to create a partitioned table. Which option is better depends on how the table will be populated and how many partitions will be created. Populating a heap and then building the clustered index is likely to provide better performance than loading into an already indexed table. Additionally, when you have multiple CPUs you can load data into the table from parallel BULK INSERTs, and then also build the indexes in parallel. For the **Orders** table, create the table normally and then load the existing data through INSERT SELECT statements that pull data from the **AdventureWorks** sample database. To create the **Orders** table as a partitioned table, specify the partition scheme in the ON clause of the table. The **Orders** table is created with the following syntax:

Copy Code

```
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 it must include the **OrderDate**, the **OrderDetails** table is created with the following syntax:

Copy Code

```
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 for loading the data is handled through two INSERT statements. These statements use the new **AdventureWorks** database from which data is copied. Install the **AdventureWorks** sample database to copy this data:

Copy Code

```
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, and the minimum and maximum **OrderDate**. A partition that does not contain rows will not be returned by this query.

Copy Code

```
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, after the tables have been populated, you can build the clustered indexes. In this case, the clustered index will be defined on the primary key because a partitioning key identifies both tables (for **OrderDetails**, add the **LineNumber** to the index for uniqueness). 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.

Copy Code

```
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:

Copy Code

```
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 provides 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. Regardless of how the partitions are joined, SQL Server always evaluates whether or not some level of partition elimination is possible.

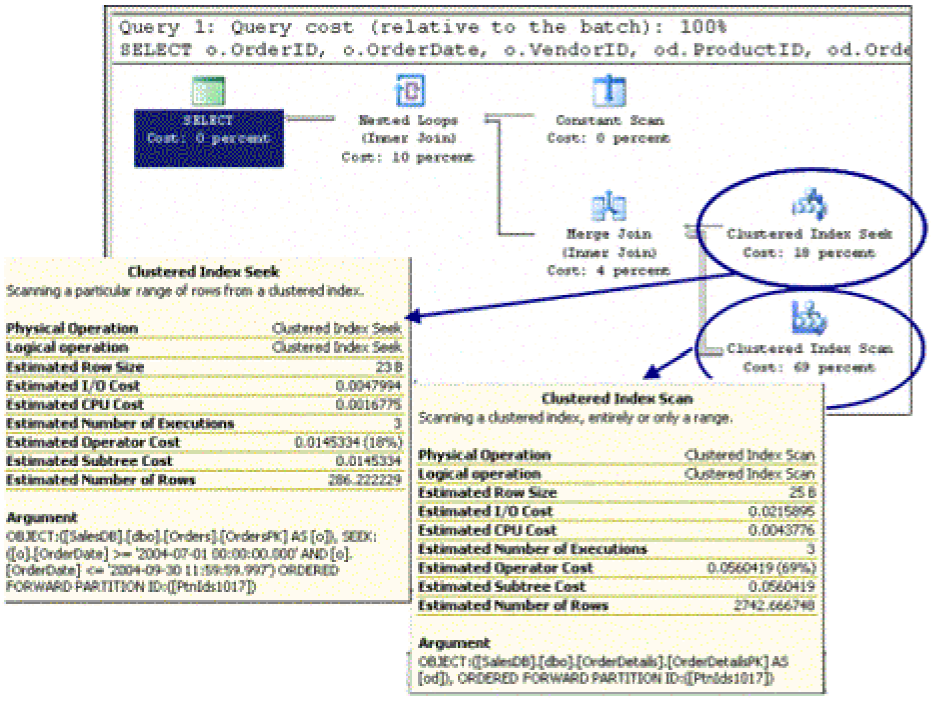
Partition Elimination

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 only from the third quarter. 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 are interested in the trends related to **Products** (the quantities ordered and their order dates) for the third quarter. To ensure that aligned partitioned tables benefit from partition elimination when joined, you must specify 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:

Copy Code

```
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
```

As shown in Figure 14, there are some key elements to look for when reviewing 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, one quarter, or three months worth of data is visible. Each month has its own partition and there are three executes in looking up this data: one for each table.



```
Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, o.VendorID, od.ProductID, od.Orde
```

Clustered Index Seek
Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical operation	Clustered Index Seek
Estimated Row Size	23 B
Estimated I/O Cost	0.0047994
Estimated CPU Cost	0.0016775
Estimated Number of Executions	3
Estimated Operator Cost	0.0145334 (18%)
Estimated Subtree Cost	0.0145334
Estimated Number of Rows	286.222229

Argument
OBJECT:([SalesDB].[dbo].[Orders].[OrdersPK] AS [o]), SEEK:([o].[OrderDate] >= '2004-07-01 00:00:00.000' AND [o].[OrderDate] <= '2004-09-30 11:59:59.997') ORDERED FORWARD PARTITION ID:([PtnIds1017])

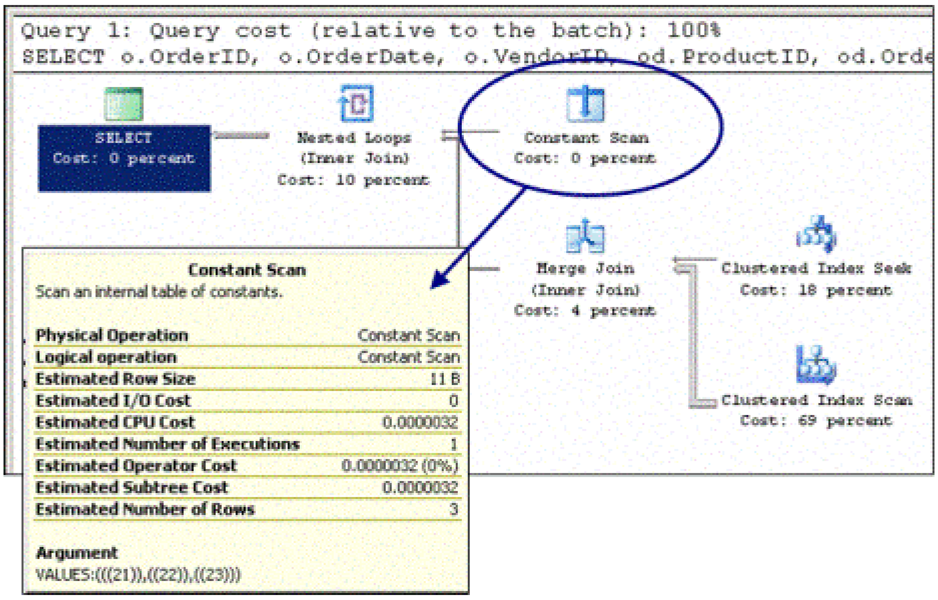
Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical operation	Clustered Index Scan
Estimated Row Size	25 B
Estimated I/O Cost	0.0215895
Estimated CPU Cost	0.0043776
Estimated Number of Executions	3
Estimated Operator Cost	0.0560419 (69%)
Estimated Subtree Cost	0.0560419
Estimated Number of Rows	2742.666748

Argument
OBJECT:([SalesDB].[dbo].[OrderDetails].[OrderDetailsPK] AS [od]), ORDERED FORWARD PARTITION ID:([PtnIds1017])

[http://msdn2.microsoft.com/en-us/library/ms345146.sql2k5partition_14(en-us,sql.90).gif]
Figure 14. Number of executes

As 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 to see what is being evaluated. You may wonder where the "PtnIds1017" expression comes from. This is a logical representation of the partitions accessed in this query. 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.



Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, o.VendorID, od.ProductID, od.Orde

Constant Scan
Scan an internal table of constants.

Physical Operation	Constant Scan
Logical operation	Constant Scan
Estimated Row Size	11 B
Estimated I/O Cost	0
Estimated CPU Cost	0.0000032
Estimated Number of Executions	1
Estimated Operator Cost	0.0000032 (0%)
Estimated Subtree Cost	0.0000032
Estimated Number of Rows	3

Argument
VALUES:(((21)),((22)),((23)))

Figure 15. Partition Elimination

To verify what data exists on each of the partitions, and only those partitions, use a slightly modified version of the query used earlier to access the new built-in system functions for partitions:

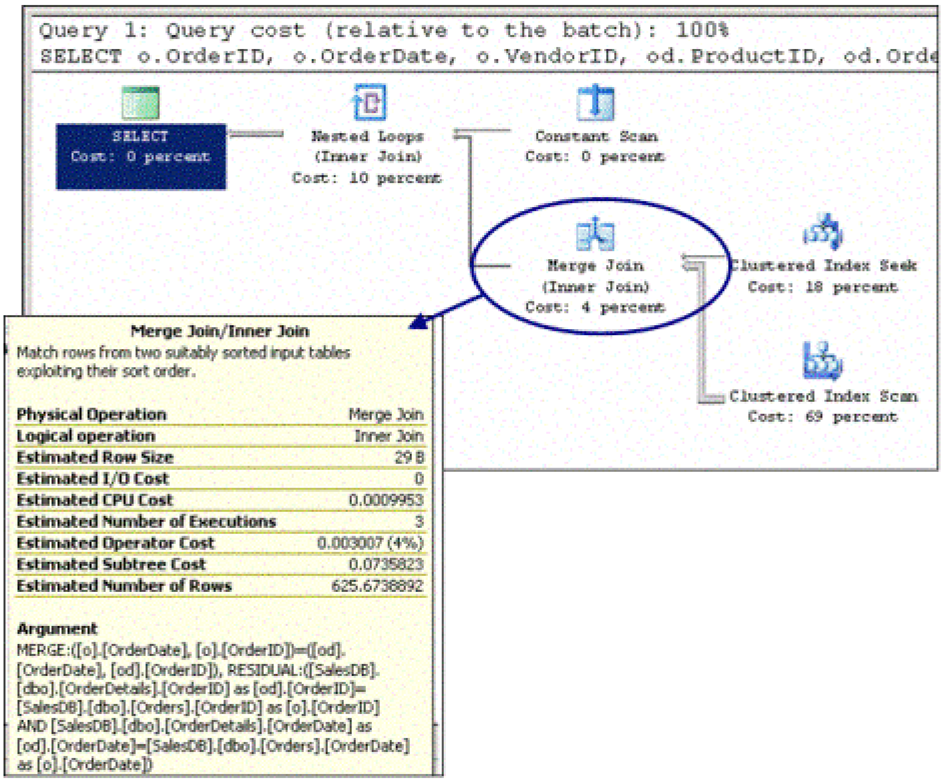
Copy Code

```
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
```

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 reviewing the number of executes for each table access, 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.



```
Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, o.VendorID, od.ProductID, od.Orde
```

Merge Join/Inner Join

Match rows from two suitably sorted input tables exploiting their sort order.

Physical Operation	Merge Join
Logical operation	Inner Join
Estimated Row Size	29 B
Estimated I/O Cost	0
Estimated CPU Cost	0.0009953
Estimated Number of Executions	3
Estimated Operator Cost	0.003007 (4%)
Estimated Subtree Cost	0.0735823
Estimated Number of Rows	625.6738892

Argument

MERGE:([o].[OrderDate], [o].[OrderID])=([od].[OrderDate], [od].[OrderID]), RESIDUAL:([SalesDB].[dbo].[OrderDetails].[OrderID] as [od].[OrderID]=[SalesDB].[dbo].[Orders].[OrderID] as [o].[OrderID] AND [SalesDB].[dbo].[OrderDetails].[OrderDate] as [od].[OrderDate]=[SalesDB].[dbo].[Orders].[OrderDate] as [o].[OrderDate])

Figure 16. Joining partitioned tables

In Figure 16, notice there's an additional nested-loop join performed. It appears that this happens after the merge join but actually the partition IDs were already passed to each table seek or scan; 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

When the next month's data is available, October 2004 in this case, there is a specific sequence of steps to follow to use the existing filegroups and to switch the data in and out. In this sales scenario, the data currently in FG1 is data from October 2002. Now that the data for October 2004 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. Specifically, the new table (the source or destination, i.e., a partition in disguise) must be created on the same filegroup that is being switched in or out. If you plan to continue to use the same filegroups (FG1 in this case), then you need to determine how to handle the space and archiving requirements. To minimize the amount of time where the table does not have two complete years of data, and if 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 month and the month to be archived, then you will need to switch the old partition out (and remove or drop it) first.

Regardless, archiving should be easy and is probably already being done. A good practice for archiving is to back up the filegroup immediately after the new partition is loaded and switched in, not just before you plan to switch it out. For example, if there were a failure on the RAID array, the filegroup could be restored rather than having to rebuild or 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. There are numerous backup strategies that can be implemented in SQL Server 2005, and many offer better granularities for backup and restore. Because so much of the data is not changing, you can back up the individual filegroups after they are loaded. In fact, this should become part of your rolling partition strategy. For more information, see "File and Filegroup Backups" in "Administering SQL Server" in SQL Server Books Online [http://www.microsoft.com/sql/techinfo/productdoc/2000/books.asp] .

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, the process is the same each month. By using dynamic SQL execution, you can easily automate this process, using these steps:

- 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.
- Back up the filegroup.

The syntax and best practices for each step are detailed in the following sections, as well as notes to help you automate this process through dynamic SQL execution.

Manage the staging table for the partition that will be switched IN

1. Create the staging table, which is a future partition in disguise. This staging table must have a constraint that restricts its data to only valid data for the partition to be created. For better performance, 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.

   Copy Code

   ```
   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 is easy to create, as it will always be the most current month. Depending on when the process runs, detecting the month is easy with built-in functions such as DATENAME(m, getdate()). Because the table's structure must match the existing table, the primary change for each month is the table name. However, you could use the same name each month, as the 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 because 59.999 rounds up to .000, which is the first day of the next month. You can subtract 2 or 3 milliseconds because -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:

   Copy Code

   ```
   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 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.

Copy Code

```
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, 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 should consider loading them in parallel. A table that keeps track of which files were loaded and where the files are located helps control this process. You can create an 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.

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

   Copy Code

   ```
   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.

   Copy Code

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

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.

   Copy Code

```
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).

Copy Code

```
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.

Copy Code

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

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

Copy Code

```
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. Specifically, 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 which will be the next partition to use for a split.

Copy Code

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

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

Copy Code

```
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), as a best practice, continue to extend 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.

Copy Code

```
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.

Copy Code

```
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.

Copy Code

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

Back up the filegroup

What you choose to back up 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 backup strategy based on a full database has been chosen, then a full-database backup or a differential backup can be performed.

Copy Code

```
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 if 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 have 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 five partitions, specify only four in the partition function. The values do not need to be ordered (SQL Server will order them internally) but the most logical way to get the correct number of partitions is to order the partition values and leave off the highest value for the last partition (when defined as a LEFT partition function) or to order the partition values and start with the second lowest 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 has the same parameters yet exist on different drives and have the specific name of the country partition.

Copy Code

```
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, which will specify only four partitions using LEFT for the boundary condition. In this case, the list will include all countries except for the UK, because it falls last in the alphabetical list.

Copy Code

```
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 listed in alphabetical order. All five filegroups must be specified within the partitioning scheme's syntax.

Copy Code

```
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.

Copy Code

```
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
```

While range partitions are defined as supporting only ranges they also offer a way to perform other types of partitions, such as list partitions.

Summary

SQL Server 2005 offers a way to easily and consistently manage large tables and indexes through partitioning, which allows you to manage subsets of your data outside of the active table. This provides simplified management, increased performance, and abstracted application logic, as 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 solution that will still yield performance gains.

Scripts From This Paper

The scripts used in the code samples for this whitepaper can be found in the SQLServer2005PartitionedTables.zip file. Following are descriptions of each file in the zip file.

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 that demonstrate 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 of 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.