Microsoft®

Microsoft® SQL Server® 2008

SQL Server Technical Article

Partitioned Table and Index Strategies Using SQL Server 2008

Writer: Ron Talmage, Solid Quality Mentors

Technical Reviewer: Denny Lee, Wey Guy, Kevin Cox, Lubor Kollar, Susan Price – Microsoft

Greg Low, Herbert Albert - Solid Quality Mentors

Published: March 2009

Applies to: SQL Server 2008

Summary: Table partitioning can make very large tables and indexes easier to manage, and improve the performance of appropriately filtered queries. This paper contains a thorough coverage of strategies for partitioning tables, including the required partition functions and partition schemes. The benefits of partition elimination, partition parallelism, and index partition maintenance are also explored. These strategies are applied to various partitioning scenarios such as the sliding window scenario, data archiving, and partition consolidation and movement.

Introduction

When a database table grows in size to the hundreds of gigabytes or more, it can become more difficult to load new data, remove old data, and maintain indexes. Just the sheer size of the table causes such operations to take much longer. Even the data that must be loaded or removed can be very sizable, making INSERT and DELETE operations on the table impractical. The Microsoft® SQL Server® 2008 database software provides table partitioning to make such operations more manageable.

Partitioning a large table divides the table and its indexes into smaller partitions, so that maintenance operations can be applied on a partition-by-partition basis, rather than on the entire table. In addition, the SQL Server optimizer can direct properly filtered queries to appropriate partitions rather than the entire table.

This paper covers strategies and best practices for using partitioned tables and indexes in SQL Server 2008. It is intended for database architects, developers, and administrators of both data warehouse and OLTP systems, and the material is presented at an intermediate to advanced

level. For an introduction to partitioned tables, see "Partitioned Table and Index Concepts" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms190787.aspx.

Table Partitioning Overview

In SQL Server, there are two ways to partition. You can manually subdivide a large table's data into multiple physical tables, or you can use SQL Server's table partitioning feature to partition a single table. There are two primary ways to partition data into multiple tables:

- One approach to partitioning data across multiple tables is called *horizontal partitioning*, where selected subsets of rows are placed in different tables. When a view is created over all the tables, and queries directed to the view, the result is a partitioned view. In a partitioned view, you have to manually apply the required constraints and operations, and maintenance can be complex and time-consuming. (For more information, see "Types of Views" in SQL Server 2008 Books Online at http://technet.microsoft.com/en-us/library/ms190426.aspx. For a comparison of partitioned views and table partitioning, see "Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server" at http://technet.microsoft.com/en-us/library/cc966457.aspx.)

- Another method is called *vertical partitioning*, where the columns of a very wide table are spread across multiple tables containing distinct subsets of the columns with the same number of rows. The result is multiple tables containing the same number of rows but different columns, usually with the same primary key column in each table. Often a view is defined across the multiple tables and queries directed against the view. SQL Server does not provide built-in support for vertical partitioning, but the new sparse columns feature of SQL Server 2008 can be a better solution for tables that require large numbers of columns. (For more information, see "Using Sparse Columns" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/cc280604.aspx.)

SQL Server's table partitioning differs from the above two approaches by partitioning a single table: Multiple physical tables are no longer involved. When a table is created as a partitioned table, SQL Server automatically places the table's rows in the correct partition, and SQL Server maintains the partitions behind the scenes. You can then perform maintenance operations on individual partitions, and properly filtered queries will access only the correct partitions. But it is still one table as far as SQL Server is concerned.

A Table Partitioning Example

SQL Server's table partitioning is designed to make loading, aging, and other maintenance operations on large tables easier, as well as improve performance of properly filtered queries using partition elimination. Table partitioning was introduced in SQL Server 2005 Enterprise Edition and enhanced in SQL Server 2008 Enterprise. Every table in SQL Server 2005 and SQL Server 2008 may be looked upon as partitioned, in the sense that each table is given a default partition in the sys.partitions catalog view. However, a table is not truly partitioned unless it is created using a partition scheme and function, which ordinary database tables by default do not have.

A partitioned table is a unique kind of table in SQL Server. It depends on two pre-existing objects, the partition function and partition scheme, two objects that are used only for partitioned tables and indexes. Their dependency relationship is shown in Figure 1.
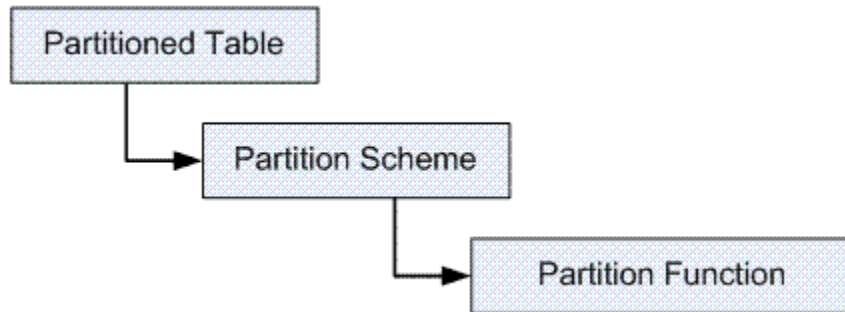
Figure 1. Partitioned table dependencies

A partitioned table has a column identified as a partitioning column, and that column is referenced when the table is created. The partitioned table must be created on a partition scheme, which defines the filegroup storage locations for the partitioned table. The partition scheme in turn depends on a previously created partition function that defines the number of partitions the table will have and how the boundaries of the partitions are defined. After all these pieces are in place, it is possible to use metadata-only operations to load and remove data from the partitioned table almost immediately.

A brief example can show how this works. This example is a simplified version of what is called a 'sliding window' scenario, which will be discussed more extensively later in this paper. We start with a partitioned table P that is created with four partitions, with only the middle two containing data, and partitioned on a key in ascending order by time. The table is partitioned on a time-sensitive column. Our goal is to load a new partition with new data and then remove the oldest partition's data.

You can see how this works in two steps. Figure 2 shows the first step, which is to use SPLIT to divide the empty partition on the leading edge of the table.
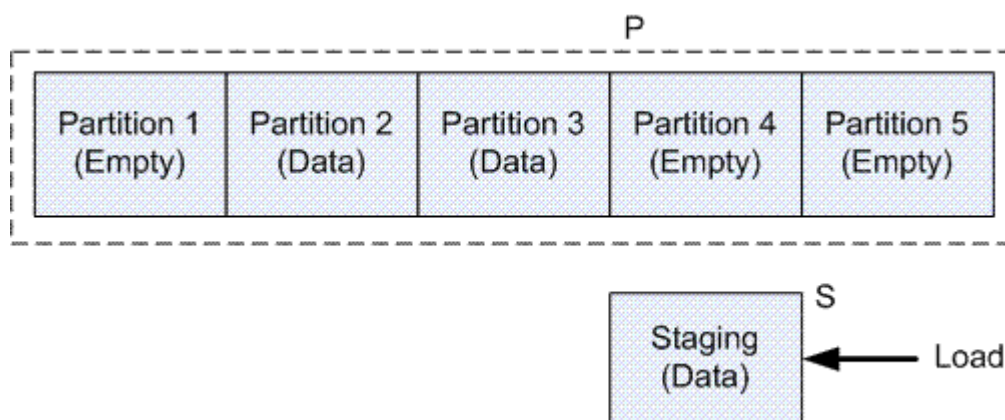

Figure 2. Before: for loading new data, a new empty partition is created and a staging table loaded

In this process, you should always maintain an empty partition on both ends of the table. For loading new data, the first step is to create a new empty partition to the right in table P. The table's partition function is modified using ALTER PARTITION FUNCTION with the SPLIT

3

option. By splitting the empty partition 4 into two partitions, 4 and 5, you ensure that SQL Server does not have to scan any partition's rows to see where they belong, and you avoid any data movement. Because the boundary between partitions 3 and 4 is later than any data in the table, there will be no data in partition 4. (If you split a nonempty partition in such a way that data will end up in either partition, SQL Server will have to move data to appropriate partition.)

Then a staging table S is loaded with new data, and appropriate indexes and a constraint are created on S to match the indexes and constraint for partition 4. This is the only time-consuming process and does not affect the partitioned table.

The next step is to use ALTER TABLE with the SWITCH option to exchange the target empty partition with the staging data, as shown in Figure 3.

P

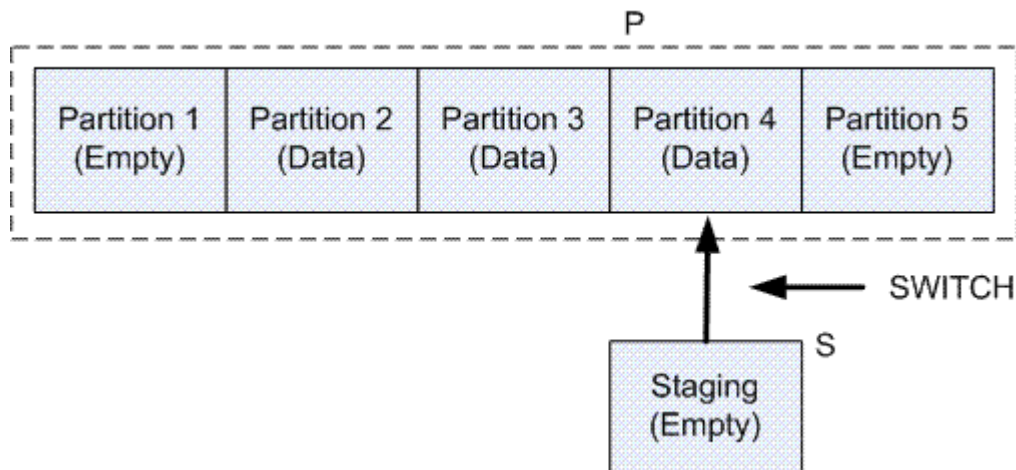| Partition 1 (Empty)	Partition 2 (Data)	Partition 3 (Data)	Partition 4 (Data)	Partition 5 (Empty)

SWITCH

S

Staging (Empty)

Figure 3. After: In the second step in the load process, the staging table data is switched or exchanged with the new partition

The actual switch takes place almost immediately, because it is simply a metadata operation. The end result is that now partition 4 has the data, and the staging table is empty.

What's key to note here is that because both creating the new empty partition and switching the data are metadata-only operations, they can take place almost immediately and without the table being offline. However, other activity might block the command and slow the operations: See the note below. The major time-consuming operation is loading the staging table.

Aging out old data is just the reverse of this process. Figure 4 shows the same table P with an empty staging table S on the same filegroup as partition 2. You are now ready to age out the oldest data in the table.

P

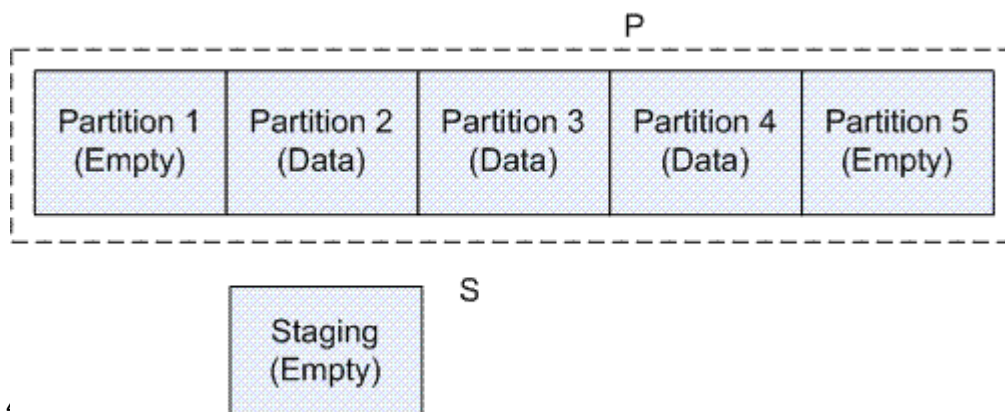| Partition 1 (Empty)	Partition 2 (Data)	Partition 3 (Data)	Partition 4 (Data)	Partition 5 (Empty)

S

Staging (Empty)

Figure 4. Before: For aging out the old data in partition 2, an empty staging table S is created on the same filegroup

The staging table S is now switched with partition 1 using the ALTER TABLE command, and the result is an empty partition 2, as shown in Figure 5.
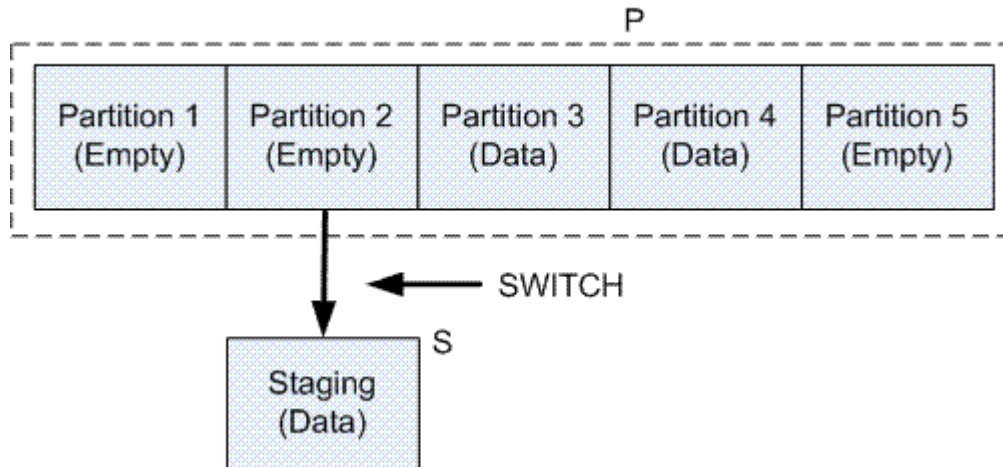

Figure 5. After: The data in partition 2 has been switched with staging table S, and the data is now in S

Again, the switch is a metadata-only operation, taking place almost immediately, and the table P remains available during that time.

Final steps include archiving off the staging data and perhaps truncating or dropping the staging table S. Then the ALTER PARTITION FUNCTION command can be used with the MERGE option to merge the two empty partitions 1 and 2, leaving the table with the original number of four partitions.

Note that in this example, empty partitions are maintained on both ends of the table. There are further considerations for actually partitioning a table, such as the choice of RANGE RIGHT or RANGE LEFT in the partition function, as well as filegroup management with the SWITCH option. These subjects are taken up in more detail in the following sections of this paper.

Here are the key points to observe from this example:

- The partitioned table stays online and remains available during the entire process, provided that the partition commands are not blocked by other activity on the table.
- Using the SWITCH option of the ALTER TABLE command involves a metadata-only operation and takes place immediately.
- Creating a new empty partition by using the SPLIT option to divide an empty partition, and removing an empty partition by merging two empty partitions, do not cause any data movement within the partitioned table and can take place almost immediately.
- The time-consuming steps are those involving loading or archiving the staging table.

This example draws attention to one of the main benefits for table partitioning: speed and manageability for data loading and archiving. It does not include details about indexes, indexed views, and the actual commands involved, or how properly filtered queries can use partition elimination against a partitioned table.

Note that other uses of the SWITCH operation are possible: For example, you can switch out a partition in the middle of a partitioned table, modify the data in the staging table, and then switch the modified data back in.

These other issues will be dealt with in the remainder of this paper. Let's first take a more in-depth look at the advantages provided by table and index partitioning.

Partitioned Table Operations and Concurrency

A schema-modification lock on the partitioned table is taken by the MERGE and SPLIT options of the ALTER PARTITION FUNCTION command and the SWITCH option of the ALTER TABLE command. This lock conflicts with the schema-stability lock taken by other DML commands (SELECT, INSERT, UPDATE, and DELETE) that may be acting on the table. As a result, even though the metadata-only operations complete in a fraction of a second, the partition commands may be blocked for a period of time by other table activity until the schema-modification lock can be obtained.

Benefits of Partitioned Tables and Indexes

SQL Server's partitioned tables and indexes offer a number of advantages when compared with partitioned views and other forms of partitioning data:

- SQL Server automatically manages the placement of data in the proper partitions.
- A partitioned table and its indexes appear as a normal database table with indexes, even though the table might have numerous partitions.
- The table can be managed at the partition and filegroup level for ease of maintenance.
- Partitioned tables support easier and faster data loading, aging, and archiving, as illustrated in the example above.
- Application queries that are properly filtered on the partition column can perform better by making use of partition elimination and parallelism.
- In cases where partitioned data will not be modified, you can mark some or most of a partitioned table's filegroups as read-only, making management of the filegroups easier.
- In SQL Server 2008, you can compress individual partitions as well as control lock escalation at a partition level.

Partitioning large tables has some challenges:

- There is a maximum of 1,000 partitions for a table.
- You must manage filegroups and file placement if you place partitions on individual filegroups.
- The metadata-only operations (SWITCH, MERGE, and SPLIT) can be blocked by other DML actions on the table at the time, until a schema-modification lock can be obtained.
- Managing date or time-based data can be complex.
- You cannot rebuild a partitioned index with the ONLINE option set to ON, because the entire table will be locked during the rebuild.
- Automating changes to partitioned tables, as in a sliding window scenario, can be difficult, but Microsoft provides some tools to assist in automating the process. (See, for

example, the Partition Management Utility on CodePlex at
http://www.codeplex.com/sql2005partitionmgmt. Also see "Two Tools to Help Automate a Sliding Window" below.)

Additional Support for Table Partitioning in SQL Server 2008

SQL Server 2008 provides a number of enhancements to partitioned tables and indexes:

- There are new wizards, the Create Partition Wizard and the Manage Partition Wizard, in SQL Server Management Studio, for creating and managing partitions and sliding windows.
- SQL Server 2008 supports switching partitions when partition-aligned indexed views are defined:
  - Management of indexed views on partitioned tables is much easier.
  - Aggregations in indexed views are preserved on existing partitions, and you only need to build new aggregates on the new partition before switching it into or out of the partitioned table.
- There are enhanced parallel query operations on partitioned tables that may provide better CPU utilization and query performance for data warehouses and multiprocessor computers.
- You can implement data compression for partitioned tables and indexes, or on specified partitions of the table.
- The **date** data type makes defining partition columns easier, because time values are not stored using **date**.
- You can configure lock escalation settings at the partition level rather than the entire table.
- Transactional replication now supports the ALTER TABLE SWITCH operation on partitioned tables for both the Publisher and the Subscriber.

Planning for Table Partitioning

In order to successfully partition a large table, you must make a number of decisions. In particular, you need to:

- Plan the partitioning:
  - Decide which table or tables can benefit from the increased manageability and availability of partitioning.
  - Decide the column or column combination upon which to base the partition.
- Specify the partition boundaries in a partition function.
- Plan on how to store the partitions in filegroups using a partition scheme.

Choosing a Table to Partition

There is no firm rule or formula that would determine when a table is large enough to be partitioned, or whether even a very large table would benefit from partitioning.

Sometimes large tables may not require partitioning, if for example the tables are not accessed much and do not require index maintenance. Further, you can place the database on multiple filegroups to gain the advantages of filegroup backups, online restore, and piecemeal restore, all without the requirement for a table to be partitioned.

7

In general, any large table has maintenance costs that exceed requirements, or that is not performing as expected due to its size, might be a candidate for table partitioning. Some conditions that might indicate a table could benefit from partitioning are:

- Index maintenance on the table is costly or time-consuming and could benefit from reindexing partitions of the table rather than the whole table at once.
- Data must be aged out of the table periodically, and the delete process is currently too slow or blocks users trying to query the table.
- New data is loaded periodically into the table, the load process is too slow or interferes with queries on the table, and the table data lends itself to a partition column based on ascending date or time.

In other words, make sure that the large table will actually benefit from partitioning; don't partition it just because it's big.

Choosing a Partition Column

After you've decided a table could benefit from being partitioned, you need to determine the partition column. The partition column stores the values upon which you want to partition the table. The partition column choice is critical, because after you have partitioned a table, choosing a different partition column will require re-creating the table, reloading all the data, and rebuilding all the indexes. Some things to note about the partition column:

- The partition column must be a single column in the table (either a single column or a computed column).
- If you have a combination of columns that form the best partition column, you can add a persisted computed column to the table that combines the values of the original columns and then partition on it. (For an example, see "The Rotating Window Scenario" below.) If you allow ad hoc queries against the partitioned table, those queries need to reference the computed column in their filters in order to take advantage of partition elimination.
- The partition column must also have an allowable data type. (For more information about the allowable data types, see "Creating Partitioned Tables and Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms188730.aspx.)
- In a clustered table, the partition column must be part of either the primary key or the clustered index. If the partition column is not part of the table's primary key, the partition column might allow NULL. Any data with a NULL in the partition column will reside in the leftmost partition.
- The partitioning column should reflect the best way to subdivide the target table. You should look for a relatively balanced distribution of data across the resulting partitions, though it may not be possible to know that in advance.
- You should also try to choose a partitioned column that will be used as a filter criterion in most of the queries run against the table. This enables partition elimination, where the query processor can eliminate inapplicable partitions from the query plan, and just access the partitions implied by the filter on the queries.

After you have identified the table and its partition column, you can then go through the steps of partitioning a table or index, which are:

1. Create or use an existing partition function that sets the correct range boundaries.
2. Create or use an existing partition scheme tied to the partition function.
3. Create the table using the partition scheme.

You use the data type of the partitioning column when defining the partition function, so effectively you must choose the partition column before defining the partition function (or deciding which partition function to use, if you have several available).

Table Partitioning Components

The partition function, partition scheme, and partitioned table or index form a dependency tree, with the partition function at the top, the partition scheme depending on the partition function, and then the partitioned table or index depending on being linked to a partition scheme. Figure 6 shows the order of creation as well as a cardinality relationship between the components.
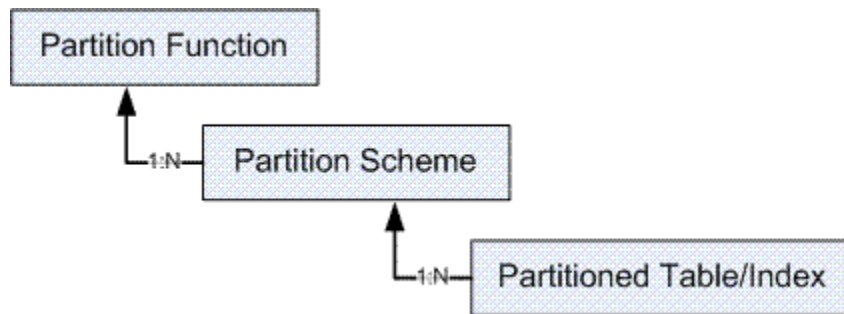

Figure 6. Dependencies of table partitioning components

In sum:

- The partition function defines the boundary values of the initial set of partitions and the data type of the partitioned column:
    - It makes no reference to any tables or disk storage.
    - It forms the basis for one or more partition schemes.
- The partition scheme maps particular partitions to filegroups:
    - A given partition scheme can be used for one or more partitioned tables, indexes, and indexed views.
- The partitioned table or index is tied to a particular partition scheme when it is created:
    - The partition table has only an indirect relationship, through the partition scheme, to the partition function.

The relationship between a partition function and a partition scheme is one-to-many, as is the relationship between a partition scheme and partitioned tables and indexes. However, in practice, because moving data in and out of a partitioned table usually requires modifying the partition function (see "The Sliding Window Scenario" below), often a partitioned table has a dedicated partition function and partition scheme.

Resources

There are a number of publicly available white papers and articles on implementing partitioned tables and indexes in SQL Server 2005:

- "Partitioned Tables and Indexes in SQL Server 2005" at http://msdn.microsoft.com/en-us/library/ms345146.aspx
- "Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server" at http://technet.microsoft.com/en-us/library/cc966457.aspx

- "Project REAL: Data Lifecycle – Partitioning" at http://technet.microsoft.com/en-us/library/cc966424.aspx
- "How to Implement an Automatic Sliding Window in a Partitioned Table on SQL Server 2005" at http://msdn.microsoft.com/en-us/library/aa964122(SQL.90).aspx

The Partition Function

In order to choose the best partition column for a table, it's critical to understand how the partition function works, because the partition function includes a data type that must be the same data type as the table's partition column. In this section, we'll look at how the partition function works for a simple integer value. In the next section, "Choosing a Partition Column," we'll take a closer look at other data types.

Partition functions are stored within the database you create them in, but they are not equivalent to user-defined functions. Instead, they are special objects used only with table partitioning.

Although user-created, partition functions differ from user-defined functions in a number of ways:

- Partition functions are not listed as database objects in the sys.all_objects or sys.objects system tables; instead, you can find them listed in sys.partition_functions.
- Partition functions are not contained by a database schema.
- Special commands must be used for creating, altering, and dropping a partition function:
    - CREATE PARTITION FUNCTION
    - ALTER PARTITION FUNCTION
    - DROP PARTITION FUNCTION
- Partition functions can be invoked interactively in Transact-SQL by using the $PARTITION function.

You create the partition function with the CREATE PARTITION FUNCTION command, specifying:

- An input parameter to the partition function declaring the data type of the partition boundaries.
- The input parameter must be compatible with subsequent table partition columns.
- A range type (either LEFT or RIGHT), which specifies how the boundary values of the partition function will be put into the resulting partitions.
- A set of constants as an initial list of boundary values. These mark the terminating points of partitions within the stated range. The boundary points must have the same data type as the input parameter. The boundary values can be modified later using ALTER PARTITION FUNCTION.

You can script out an existing partition function with Object Explorer in SQL Server Management Studio. Expand the Storage node of a database, expand the Partition Functions node, right-click the partition function name, and then click **Script Partition Function as**.

When you choose a data type for a partition function, you are committing yourself to creating partitions within the range of all possible values of that data type. Every allowable SQL Server data type determines an ordered range of possible values:

- The values are discrete and finite in number.
- The values are implicitly ordered from a minimum to a maximum.

- NULL is considered smaller than the minimum value of the data type's values.

A Single Integer Boundary Value

You specify a set of boundary values from the range of possible values of the input parameter's data type. These boundary values define a number of subranges within the overall range, called partitions. Each partition:

- Has its own portion of the overall range.
- Cannot overlap with any other partition.

Also worth noting:

- The total number of resulting partitions is the number of partition function boundary values plus one.
- Any given possible value within the boundary value range can exist in only one partition.
- Partitions are numbered starting with 1.

Let's start by dividing the range of integers, effectively cutting the range into subranges at the value 10, as shown in Figure 7.

Figure 7. A range of integers where the number 10 will become a boundary value

You now have to decide which subrange or partition the boundary value 10 should belong in: the partition to the left or the right. The partition function forces that choice using the RANGE LEFT or RANGE RIGHT clause. The partitions cannot overlap, so the boundary value 10 can only belong to one partition.

Let's start with a sample partition function that chooses RANGE LEFT.

```
CREATE PARTITION FUNCTION PF1_Left (int)
AS RANGE LEFT FOR VALUES (10);
```

Figure 8 below illustrates the boundary value 10 in the left partition, corresponding to RANGE LEFT in our PF1_Left() partition function.
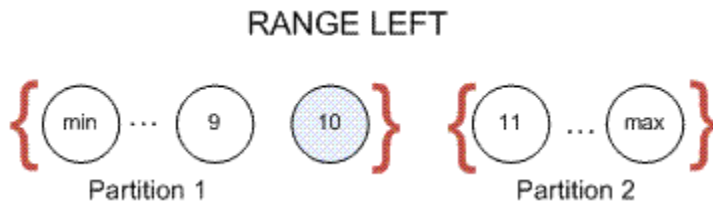

Figure 8. A single integer boundary value with RANGE LEFT

RANGE LEFT means that the boundary value falls to the partition to the left. The partitions also now have limits placed on their values. RANGE LEFT with one partition boundary illustrates that:

- For RANGE LEFT, the boundary value specifies the upper bound of its partition: 10 is the highest allowable value in partition 1.
- All values in partition 1 must to be less than or equal to the upper boundary of partition 1.
- All values in partition 2 must be greater than partition 1's upper boundary.

Next, let's create a new partition function that specifies RANGE RIGHT.

```
CREATE PARTITION FUNCTION PF1_Right (int)
AS RANGE RIGHT FOR VALUES (10);
```

Partition function PF1_Right() puts boundary value 10 into the partition on its right, as shown in Figure 9.
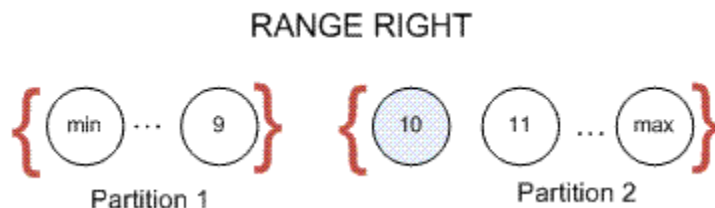

Figure 9. A single integer boundary value with RANGE RIGHT

RANGE RIGHT means that the boundary value falls to the partition on the right, but different limits now exist on the partition values. RANGE RIGHT with one boundary value shows that:

- For RANGE RIGHT each boundary value specifies the lowest value of its partition: 10 is now the lowest allowable value in partition 2.
- All values in partition 1 must be less than the lower boundary 10 of partition 2.
- All values in partition 2 must be greater than or equal to partition 2's lower boundary.

What's key is that when you choose RANGE LEFT, the boundary values all shift to the leftmost set of the partitions, and with RANGE RIGHT, they shift to the rightmost set.

For both RANGE LEFT and RANGE RIGHT, the leftmost partition has the minimum value of the data type as its lower limit, and the rightmost partition has the maximum value of the data type as its upper limit. You can place constraints to limit this later on the partitioned table, but these maximum and minimum limits always apply to the partition function's potential ranges.

Using the $PARTITION Function

You can determine where a partition function will place values by using the $PARTITION system function. $PARTITION calls the user-defined partition function, where the partition function is passed a sample data value as an input parameter. Based on the range direction and the boundary values, it returns a result indicating what partition a particular value would map to. You can use the $PARTITION function without linking it to any partition scheme or having any partitioned table using the partition function.

The following example uses $PARTITION.

```
SELECT $PARTITION.PF1_Left(10);
```

```
SELECT $PARTITION.PF1_Right(10);
```

The result shows that with PF1_Left(), 10 falls into partition 1, and with PF1_Right(), 10 falls into partition 2.

You can also use the $PARTITION function to determine what partitions a set of values from a table will fall into, using the column name of the table instead of a constant. As an example, the

following query will show the partition number and projected row count for the FactResellerSales table in the **AdventureWorksDW2008** database. It assumes a partition function named pfn_OrderDateKeyRight, which sets up monthly partitions on the table's OrderDateKey integer column, using RANGE RIGHT and running values from 200107-1 through 20040701.

```
USE AdventureWorksDW2008;

GO

SELECT $PARTITION.pfn_OrderDateKeyRight(OrderDateKey) AS PartitionNumber

     , COUNT(*) AS ProjectedCount

FROM dbo.FactResellerSales

GROUP BY $PARTITION.pfn_OrderDateKeyRight(OrderDateKey)

ORDER BY PartitionNumber;
```

For more examples, see "$PARTITION (Transact-SQL)" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms188071.aspx. You could also use the $PARTITION function to verify that a staging table's partition column values will all map to the correct partition.

Multiple Integer Boundary Values

Now let's extend the use of the partition function to multiple boundary values. When deciding on multiple boundary values for a partition function, there are several points to keep in mind:

- The boundary values in the CREATE PARTITION FUNCTION statement should be sorted from least to greatest. If they are not sorted, SQL Server will provide a warning and sort them.

- All the boundary values fall into partitions the same direction, left or right, based on the direction option of the RANGE clause.

- The boundary values must resolve to constants: You can use functions to determine the boundary values, so long as they result in constants.

- The maximum number of boundary values is 999, resulting in a maximum of 1,000 partitions.

- NULL can be a boundary value in a partition function. For RANGE LEFT, beginning a list of boundary values with NULL will place all and only data with NULLs in the partition column into partition 1. For RANGE RIGHT, beginning a list of boundary values with NULL will leave partition 1 permanently empty, and data with NULL in the partition column will fall into partition 2, along with all other values less than the next boundary value.

Because a partition scheme maps filegroups by the same order, you should keep the boundary values in an ascending order so that they will match up with the partition scheme's filegroups as you expect.

A NULL value will normally not be allowed in a partitioned table, because generally the partition column will not allow NULL, though there may be situations where it is useful. We will ignore NULL as a possible partition value in the rest of this paper.

Now consider three integer boundary values: 0, 10, and 20, with each possible range direction.

Range Direction LEFT and Integer Boundary Values

For RANGE LEFT and boundary values (0, 10, 20):

```
CREATE PARTITION FUNCTION PF2_Left (int)

AS RANGE LEFT FOR VALUES (0, 10, 20);
```

The result for this RANGE LEFT assignment is:

{min … **0}**, {1 … **10}**, {11 … **20}**, {21 … max}

The boundary values are in bold. Each boundary value falls into the partition on its left as the maximum upper value for that partition. In other words, the three boundary values serve as the upper boundaries of the leftmost three partitions out of the total four.

Note the resulting limits for RANGE LEFT and (0, 10, 20):

Partition 1: values from the data type minimum to <= 0

Partition 2: values > 0 and <= 10

Partition 3: values > 10 and <= 20

Partition 4: values > 20 and <= data type maximum

We end up grouping the integers as min-0, 1-10, 11-20, 21-max.

But it's more natural to group these integers as min through -1, 0-9, 10-19, 20-max. You can achieve this by rewriting the RANGE LEFT function to state the maximum partition boundary values of the partition.

```
CREATE PARTITION FUNCTION PF2_Left_Revised (int)

AS RANGE LEFT FOR VALUES (-1, 9, 19);
```

Now the new ranges are:

{min … **-1}**, {0 … **9}**, {10 … **19}**, {20 … max}

To make a more natural grouping with RANGE LEFT, you must explicitly state the boundary values as upper boundaries of the each partition range.

Range Direction RIGHT and Integer Boundary Values

Now let's use the same boundary values (0, 10, 20) but with RANGE RIGHT.

```
CREATE PARTITION FUNCTION PF2_Right (int)

AS RANGE RIGHT FOR VALUES (0, 10, 20);
```

Now the boundary values fall into each partition on the right: Each boundary value becomes the minimum value for that partition. The result, with boundary values in bold, is:

{min … -1}, {**0** … 9}, {**10** … 19}, {**20** … max}

Now we achieve the natural grouping of the integer values by simply specifying the lower boundaries of each partition. The resulting limits are:

Partition 1: values from the data type minimum to < 0

Partition 2: values >= 0 and < 10

Partition 3: values >= 10 and < 20

Partition 4: values >= 20 and <= data type maximum

Working with Boundary Data Types

Generally speaking, RANGE RIGHT gives more satisfying results than RANGE LEFT because we often think of ranges with the lowest value as the beginning of the range. This is especially

true for the most common types of boundary data types: integers and time-based data types such as **datetime**, **date**, and **time**.

Integer Boundary Values

The comparison of RANGE LEFT and RANGE RIGHT in the previous section shows that it is important to take the resulting partition boundary limits into account when choosing a set of boundary values.

For ranges of integers, RANGE RIGHT results in a more natural set of partition boundaries: It's more natural to think of the ranges 0-9, 10-19, 20-29, and so on than 1-10, 11-20, 21-30, and so on. This may be only a minor advantage for integers, but it becomes more significant with other data types.

There is a case where RANGE LEFT can be more natural. Suppose you want to use the integer data type to create partitions that only have one value, and the value to match the partition number. For example, you might want partition 1 to match with the boundary value 1, partition 2 with the boundary value 2, and so on. You can accomplish this using RANGE LEFT and specifying partition boundaries. For example, specifying RANGE LEFT with boundary values 1, 2, 3, and 4, with an integer data type, would result in the partitions:

{min … **1**), {**2**}, {**3**}, {**4** … max)

You would need to make sure that values of NULL, 0, and lower are not allowed by placing a constraint on the partitioned table so that partition 1 will only allow the value 1, and the last partition would only allow the upper boundary value.

datetime Boundary Values

The **datetime** data type presents some challenges because it is a composite of both date and time. For **datetime**, RANGE RIGHT keeps its advantages over RANGE LEFT. Specifying lower boundaries using RANGE RIGHT is simpler with **datetime**, as you can see from an example. The SQL Server 2005 PartitionAW.sql sample code for partitioning **AdventureWorks** partitions the Production.TransactionHistory table on a monthly basis. It uses a **datetime** data type and RANGE RIGHT to partition the table.

```
CREATE PARTITION FUNCTION [TransactionRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('10/01/2003', '11/01/2003', '12/01/2003',
              '1/01/2004', '2/01/2004', '3/01/2004', '4/01/2004',
              '5/01/2004', '6/01/2004', '7/01/2004', '8/01/2004');
```

This partition function relies on the conversion of strings into the **datetime** values that will become the lower boundary levels of each partition. So '10/01/2003' is translated as 10/01/2003 00:00:00.000, or 10/01/2003 midnight. The resulting partition constraints are:

Partition 1: values from the data type minimum to < 10/01/2003 midnight

Partition 2: values >= 10/01/2003 midnight and < 11/01/2003 midnight

Partition 3: values >= 11/01/2003 and < 12/01/2003 midnight

…

Partition 12: values >= 8/01/2004 midnight and <= data type maximum

Using RANGE LEFT would force you to explicitly state the upper boundary of each partition, which quickly becomes complex. You have to take into account how SQL Server treats fractions of seconds in the **datetime** data type. Although you can submit any fraction of a second to SQL Server, it only stores them in terms of 3 milliseconds: at .nn0, .nn3, and .nn7. If you submit the

time string '11:59:59.998' for conversion to **datetime**, SQL Server will convert the fraction of a second to .997, whereas '23:59:59.999' will be converted to midnight of the next day. Therefore, to exactly specify the same function as RANGE LEFT, you have to include the fractions of a second before midnight, adjusted for the conversion of milliseconds by SQL Server. The following code shows the same partition function as above, but now used as RANGE LEFT.

```
CREATE PARTITION FUNCTION [TransactionRangePF1_Left1] (datetime)
AS RANGE LEFT FOR VALUES ('10/01/2003 23:59:59.997',
'11/01/2003 11:59:59.997', '12/01/2003 23:59:59.997',
    '1/01/2004 11:59:59.997', '2/01/2004 23:59:59.997',
'3/01/2004 11:59:59.997', '4/01/2004 23:59:59.997',
    '5/01/2004 11:59:59.997', '6/01/2004 23:59:59.997',
'7/01/2004 11:59:59.997', '8/01/2004 23:59:59.997');
```

The resulting RANGE LEFT function is harder to read and maintain because of the necessity to specify the upper time boundary. The advantages of RANGE RIGHT stand out clearly: It's more maintainable over the long term, and you can convert the RANGE RIGHT boundaries to other date data types such as **date** or **datetime2** without worrying about rounding issues.

There is an important difference between the two resulting partitions: The RANGE RIGHT partition function will result in a partition on the left that covers all values less than the lowest stated boundary value, whereas the RANGE LEFT will result in a partition on the right that has all values greater than the highest declared boundary value. This is not really a problem, but you should keep it in mind when developing the sliding window scenario.

Note that in the RANGE LEFT version of the partition function, you can't leave off the time and just use the dates. For example, you might want to create the partition function as follows.

```
CREATE PARTITION FUNCTION [TransactionRangePF1_Left2] (datetime)
AS RANGE LEFT FOR VALUES ('10/31/2003', '11/30/2003', '12/31/2003',
                '1/31/2004', '2/29/2004', '3/31/2004', '4/30/2004',
                '5/31/2004', '6/30/2004', '7/31/2004', '8/31/2004');
```

Here the boundary values are stated just in days, and we let SQL Server convert them to midnight. But each partition will have a strange behavior. For example, 10/31/2003 00:00:00.000 will be in partition 1, but 10/31/2003 00:00:00.003 will be in partition 2. In fact, all the October values will be in partition 2 except for the very first midnight value. That extends to the rest of the function: The midnight value for each partition's upper boundary will be in the wrong partition. So you must include the time in the boundary values if you are using RANGE LEFT and want to keep each month's data in the same partition.

DATE Boundary Values

It is possible to partition on date values, either by converting a **datetime** to an integer or using the SQL Server 2008 **date** data type.

For example, in the **AdventureWorksDW2008** sample database for SQL Server 2008, the FactInternetSales table has an integer column OrderDateKey, which has been converted from **datetime** values during the staging process. Each integer value represents a day only, in an YYYYMMDD format, with no time value. As a result, a RANGE RIGHT partition function can list just the lower boundaries for each month, as in the following example.

```
CREATE PARTITION FUNCTION pfn_OrderDateKeyRight (integer)

AS RANGE RIGHT FOR VALUES (

20010701, 20010801, 20010901, 20011001, 20011101, 20011201,

20020101, 20020201, 20020301, 20020401, 20020501, 20020601,

20020701, 20020801, 20020901, 20021001, 20021101, 20021201,

20030101, 20030201, 20030301, 20030401, 20030501, 20030601,

20030701, 20030801, 20030901, 20031001, 20031101, 20031201,

20040101, 20040201, 20040301, 20040401, 20040501, 20040601,

20040701, 20040801

);
```

If you were to use the RANGE LEFT on this column, you would have to specify the upper days for each month, taking account that the final day of some months are 30, some 31, and February sometimes 28 and sometimes 29. By specifying the lower boundary value, RANGE RIGHT makes the boundary value list more manageable.

Similarly with the **date** data type (new in SQL Server 2008). Let's set the partition boundaries of the first three months in the year 2008: 20080101, 20080201, and 20080301. Again, RANGE RIGHT is more convenient.

CREATE PARTITION FUNCTION pfn_DateRight (date)

AS RANGE RIGHT FOR VALUES (

'20080101', '20080201', '20080301'

);

Because RANGE RIGHT specifies the lower boundary of each partition except the leftmost, you can simply state the beginning of each month as a boundary value.

datetime2 and datetimeoffset Boundary Values

Two additional composite datetime data types are available in SQL Server 2008: **datetime2** and **datetimeoffset**. They are similar to **datetime** for partitioning purposes.

Like the **date** data type, **datetime2** expands the date range to January 1, year 1, and like **datetime**, **datetime2** keeps fractional seconds, except that the fractional seconds are variable, potentially with a granularity of 100 nanoseconds. The default time fractional resolution of **datetime2** is 7 decimal places, the same as **datetime2**(7).

datetimeoffset is like **datetime2** in its date range and in fractional seconds. So for both **datetime2** and **datetimeoffset**, there are two considerations:

- When you create a partition function using **datetime2**, the resulting partition table column must use the same time fractional resolution.

- Again, RANGE RIGHT will be more convenient for forming boundaries based on date using **datetime2**.

datetimeoffset includes a time zone offset that can affect how a table's partition column value will resolve to a partition. You can use the $PARTITION function to determine what partition a particular value will resolve to. For example, if you define a partition function using **datetimeoffset** as RANGE RIGHT with boundaries ('20080101', '20080201'), you can use $PARTITION to determine that '20080101 00:00:00' will resolve to partition 2, but '20080101

00:00:00 +01:00' will resolve to partition 1, because it is one hour earlier than midnight 20080101 for the SQL Server instance.

Cautions Regarding date and datetime Strings

Be careful when using any of the date data types in a partition function. The strings that are listed in the boundary values assume that us_english is the session language. Do not use a style of specifying date strings that might be interpreted in different ways depending upon the language. For example, suppose you define a partition function called PF3_Right_s(), using the language setting as Spanish, with boundary values for January, February, and March as follows.

```
SET LANGUAGE Spanish;

GO

IF EXISTS(SELECT * FROM sys.partition_functions WHERE name =
'PF3_Date_Right_sp')

    DROP PARTITION FUNCTION PF3_Date_Right_sp;

GO

CREATE PARTITION FUNCTION PF3_Date_Right_sp (date)

AS RANGE RIGHT FOR VALUES ('01/01/2009',' 02/01/2009', '03/01/2009');

GO

SELECT $PARTITION.PF3_Date_Right_sp('02/01/2009');

SELECT $PARTITION.PF3_Date_Right_sp('03/01/2009');

SELECT CAST('02/01/2009' AS Date);

SELECT CAST('03/01/2009' AS Date);
```

Even though the CAST function shows the Spanish interpretation of '02/01/2009' as January 2, CREATE PARTITION FUNCTION and $PARTITION both treat it as February 1. A similar issue arises with the **datetime** data type and the SET DATEFORMAT command. The SQL Server default format is mdy (month, day, year). If you set the date format to dmy, the CAST and CONVERT functions will interpret '2009-02-01' as January 1, but the partition functions will not.

> **Note** For safety, always specify date strings for all date data types so that they do not depend on language-or date format, such as YYYYMMDD, in your partition functions. (For more information, see "CREATE PARTITION FUNCTION" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms187802.aspx.)

TIME Boundary Values

You can partition on the **time** data type, which also carries fractions of a second and therefore can also benefit from RANGE RIGHT. Suppose, for example, you have time values that are accurate to two hundredths of a second, but you want to partition by hour. Then you could create a partition function as follows.

```
CREATE PARTITION FUNCTION pfn_TimeRight (TIME(2))

AS RANGE RIGHT FOR VALUES (

'00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00', '05:00:00',

'06:00:00', '07:00:00', '08:00:00', '09:00:00', '10:00:00', '11:00:00',
```

```
'12:00:00', '13:00:00', '14:00:00', '15:00:00', '16:00:00', '17:00:00',
'18:00:00', '19:00:00', '20:00:00', '21:00:00', '22:00:00', '23:00:00'
);
```

Again, RANGE RIGHT makes the boundary value declaration much easier.

Other Data Types as Boundary Values

Integers and time-and-date data types are the most natural methods of partitioning database data, but you can also use other data types.

You might consider partitioning on a character (string) data type, but it can be tricky. The input parameter of the partition function must exactly match the data type of the table column (for example, if the data type of the table's partition column is **varchar(20)**, the partition function's input parameter must also be **varchar(20)**). Further, the ranges of the character data in the resulting partitions will be based on the collation and sort order of the database. If you supply boundary values 'A', 'B', and 'C' for example, strings beginning with lower case letters will appear in the last partition and those beginning with a numeric character will appear in the first. For an example of partitioning on a character column, see "CREATE PARTITION FUNCTION" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms187802.aspx.

Other data types, such as **numeric**, **decimal**, and **money**, can also be partitioned, but certainly by far the most useful data types for partitioning are integers and the time-based data types.

> **Note** Partitioning a FILESTREAM table may require partitioning on a ROWGUID column. For more information, see "Partitioning a Table with a FILESTREAM Column" in "Additional Partitioning Support in SQL Server 2008" below.

Altering a Partition Function: SPLIT and MERGE

Partition functions are not static: They can change over time, and their changes propagate through the partition scheme to the partitioned table. You can alter a partition function to change the boundary value list one value at a time, a method that makes it possible to add and remove new partitions. (You cannot change the partition function's data type or range direction, though, just the boundary values.)

Using SPLIT

In this section we look at the SPLIT option by itself. In the section on the partitioning scheme below, you'll find out that it is also necessary to plan for the next-used filegroup when using SPLIT with a partition function used by a partition scheme.

You can add a single boundary value using the SPLIT clause of ALTER PARTITION FUNCTION:

- The new boundary value must be in the allowed range and distinct from existing boundary values.
- Adding a boundary value has the effect of splitting the table partition that the new boundary value falls in.
- You add the new boundary value using the SPLIT keyword in ALTER PARTITION FUNCTION.
- The new boundary value obeys the RANGE direction specified in CREATE PARTITION FUNCTION.

19

- The ALTER PARTITION FUNCTION SPLIT and MERGE options can reference variables or functions as boundary values, as long as they resolve to a constant of the same data type as the input parameter.
- If the partition function is used by a partitioned table and SPLIT results in partitions where both will contain data, SQL Server will move the data to the new partition. This data movement will cause transaction log growth due to inserts and deletes.

For example, recall the PF2_Right() partition function (which currently has boundary values at 0, 10, 20) and therefore has a set of partitions:

{min … -1), {**0** … 9}, {**10** … 19}, {**20** … max}

Now let's add a new boundary value to split partition 3 at the value 15.

```
ALTER PARTITION FUNCTION PF2_RIGHT()
SPLIT RANGE (15);
```

The result now is four boundary values (0, 10, 15, and 20) and therefore five partitions.

{min … -1}, {**0** … 9}, {**10** … 14}, {**15** … 19}, {**20** … max}

The new partition beginning with 15 is now partition 4, and the old partition 4 (20 … max) gets renumbered as partition 5.

In the above example, inserting the boundary value 15 will move any data with values from 15 through 19 into the new partition 4, which contains values 15 through 19.

Using MERGE

In this section we look at the MERGE option by itself. In the section on the partitioning scheme below, you'll find out which filegroup is freed up from the partitioned table after a MERGE operation.

Similarly, you can remove a single boundary value using MERGE:

- Removing a boundary value has the effect of merging two neighboring partitions into a single new partition.
- Removing a boundary value reduces the total number of boundary values by one, and therefore the number of partitions by one also.

For example, suppose you remove the boundary value 10 in the PF2_Right() partition function (which currently has boundary values at 0, 10, 15, 20) and the following partitions:

{min … -1}, {**0** … 9}, {**10** … 14}, {**15** … 19}, {**20** … max}

The following command will remove the partition with lower boundary 10.

```
ALTER PARTITION FUNCTION PF2_RIGHT()
MERGE RANGE (10);
```

The resulting partitions are:

{min … -1}, {**0** … 14}, {**15** … 19}, {**20** … max}

You can check the result by using the $PARTITION function with sample values, or by querying the metadata using the queries in the Appendix. We now have four partitions, because the boundary value for partition 3 was removed. The old partitions 4 and 5 get renumbered as partitions 3 and 4 respectively.

> **Note**   If you remove all the boundary values from a partition function, the result will be a partition function with only one partition (partition 1). Because the partition function has no boundary values, it will have no rows in the sys.partition_range_values catalog view.

For information about how SPLIT and MERGE work when the partition function is used with a partition scheme and filegroups, see "Managing Filegroups with SPLIT and MERGE" below.

Avoiding Data Movement

Both SPLIT and MERGE have effects that follow though to all the partition function's partition schemas and partitioned tables. If you use SPLIT to divide an empty partition into two empty partitions, SPLIT will be a metadata-only operation and you will avoid data movement. Similarly, if you use MERGE to combine two empty partitions, the MERGE operation will also avoid data movement.

However, if you are using SPLIT and MERGE with nonempty partitions, data movement will occur. For example, a MERGE operation on two nonempty partitions will cause the data from the one partition to be inserted into the remaining partition and then deleted from the old partition. The resulting data movement will cause the transaction log to grow.

The Partition Scheme

The next step after defining a partition function is to map the partition function to one or more partition schemes. The partition scheme defines how any resulting partitions will be stored on filegroups. You list the filegroups for a partition scheme in the same order that you want to have them map to the partitions defined by the partition function. Creating a partition scheme assumes that you have already created the filegroups.

Assuming four filegroups have been created, the following code defines filegroups for the PF2_Right() function defined above, with the three boundary values (0, 10, 20).

```
CREATE PARTITION SCHEME PS2_Right

AS PARTITION PF2_Right

TO (Filegroup1, Filegroup2, Filegroup3, Filegroup4);
```

SQL Server accordingly assigns the first partition to Filegroup1, the second to Filegroup2, and so on, for all four partitions.

If you want to place all the partitions on a single filegroup, you state it using the ALL keyword, and you can place all partitions on the database's primary filegroup using the PRIMARY keyword. (For more information, see "CREATE PARTITION SCHEME" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms179854.aspx.)

> **Note**   To use the filegroups with a partitioned table, you must also add files for each filegroup. For faster data loading, it is usually recommended to assign more than one file per filegroup. For simply defining a partition scheme, the files are not required, but they will be when it comes time to partition a table.

You must provide a sufficient number of filegroups to handle all of the partition function's partitions. If you supply more filegroups than there are partitions defined in the partition function, SQL Server will mark the first filegroup listed after the existing partitions are mapped as NEXT USED, that is, the next filegroup to use if the number of partitions increases, and ignore the rest. SQL Server only keeps track of the existing mapped filegroups and the single filegroup designated as NEXT USED.

Like the partition function, a partition scheme is a metadata object: No actual table partitions exist until a table is created referencing the partition scheme.

You can generate a script of a partition scheme using Object Explorer in SQL Server Management Studio. Expand the Storage node of a database, expand the Partition Schemes node, and then right-click the partition scheme name.

Altering a Partition Scheme

After you've created a partition scheme, there's only one change you can make to it: You can adjust which filegroup will be considered the next used. For example, the following code makes Filegroup5 the next to be used.

```
ALTER PARTITION SCHEME PS2_Right NEXT USED Filegroup5;
```

To remove all potential filegroups from being marked as next used, just reissue the ALTER PARTITION SCHEME command with no filegroups.

```
ALTER PARTITION SCHEME PS2_Right NEXT USED;
```

Marking a filegroup as NEXT USED means that if a partition is split by adding a new boundary value using ALTER PARTITION FUNCTION, the new partition will be assigned to the filegroup marked as NEXT USED.

Using SPLIT with Partition Schemes

If you modify a partition function to add or remove a boundary value with SPLIT or MERGE, SQL Server automatically adjusts any partition scheme's filegroups that use that partition function.

To add a new boundary value to the partition function used a partition scheme, you must have a filegroup that has been marked as NEXT USED. In our example PS2_Right partition scheme, the partition function PF2_Right() has three boundary values (0, 10, 20). If the partition scheme PS2_Right has Filegroup5 marked as NEXT USED, adding a new boundary value will cause any new partition to use Filegroup5. In general, for RANGE RIGHT, when you split a partition, the result is a new partition with the new boundary value as its lowest value. SQL Server places the new partition on the next used filegroup, and then it shifts the partition numbers of the remaining partitions up by 1.

Suppose you start with the following:

{min … -1},	{**0** … 9},	{**10** … 19},	{**20** … max}	
Filegroup1	Filegroup2	Filegroup3	Filegroup4	

Adding a SPLIT at 25 will produce:

{min … -1},	{**0** … 9},	{**10** … 19},	{**20** … 24},	{**25** … max}
Filegroup1	Filegroup2	Filegroup3	Filegroup4	Fillegroup5

Adding a SPLIT instead at 15 results in:

{min … -1},	{**0** … 9},	{**10** … 14},	{**15** … 19},	{**20** … max}
Filegroup1	Filegroup2	Filegroup3	Filegroup5	Fillegroup4

Therefore to coordinate partitions with filegroups, it is critical to know which filegroup you have designated as the NEXT USED.

In a RANGE LEFT partition scheme, the boundary values represent the highest value in a partition. So when you split a RANGE LEFT partition, the new partition uses the new boundary value as its highest and resides on the NEXT USED filegroup.

If a table exists using the partition scheme, and you perform a SPLIT or a MERGE operation on the data, SQL Server may have to redistribute the data in the table, resulting in costly data

movement. Rows in the table with partition column values that fall into the new partition will be moved to the appropriate new filegroup. As a result, it may be more efficient to switch out a partition's data to a staging table, split an empty partition, and then distribute the data to two appropriate staging tables before switching it back in. A similar consideration for MERGE is that if you switch out a partition and then merge the partition with another, you cannot switch the data back in because the target partition will not be empty. It's important to keep the staging table coordinated with the partitions. For more information about the SWITCH option, see "Partitioned Table and Indexes" below.

Using MERGE with Partition Schemes

If you remove a boundary value using the MERGE option, a partition is removed from the partition function, and therefore one less filegroup is required. The filegroup that is removed is the filegroup where the removed boundary point resided.

When two partitions are merged in RANGE RIGHT, the new lowest value of the range is the one from the lower partition, and the new partition remains with that filegroup. The data from the filegroup on the right is moved to the filegroup on the left, and all remaining higher partition numbers are decremented by 1. The filegroup that was removed from the partition scheme will be marked as the NEXT USED if no other filegroup was already marked as such.

For example, considering four partitions with four filegroups, using RANGE RIGHT:

{min … -1),	{**0** … 9},	{**10** … 19},	{**20** … max}
Filegroup1	Filegroup2	Filegroup3	Filegroup4
Partition 1	Partition 2	Partition 3	Partition 4

If you merge partitions 2 and 3 by removing the boundary value 10, the result will be:

{min … -1),	{**0** … 19},	{**20** … max}
Filegroup1	Filegroup2	Fillegroup4
Partition 1	Partition 2	Partition 3

SQL Server will move all data from the old partition 3 on Filegroup3 into the **leftward** Filegroup2 with the merged partition 2. This data movement will cause transaction log growth due to inserts and deletes. Filegroup3 will be the default NEXT USED filegroup.

RANGE LEFT behaves similarly, but any data from the removed filegroup will move to the right. For example, considering four partitions with four filegroups, using RANGE LEFT:

{min … **0**),	{1 … **10**},	{11 … **20**},	{21 … max}
Filegroup1	Filegroup2	Filegroup3	Filegroup4
Partition 1	Partition 2	Partition 3	Partition 4

If you MERGE partitions 2 and 3 by removing the boundary value 10, the result will be:

{min … **0**),	{1 … **20**},	{21 … max}
Filegroup1	Filegroup3	Fillegroup4
Partition 1	Partition 2	Partition 3

SQL Server will move all data from the old partition 2 on Filegroup2 and move it into the **rightward** merged partition 2 and Filegroup3. This data movement will cause transaction log growth due to inserts and deletes. Filegroup3 will be the default NEXT USED filegroup. This data movement will cause transaction log growth due to inserts and deletes.

Just as with SPLIT, if a table exists using the partition scheme, rows in the table with partition column values in the old partition will be moved to the merged filegroup. To minimize data movement, you should merge an empty partition with a neighboring partition.

Managing Filegroups with SPLIT and MERGE

You can use multiple filegroups with partitioned tables in order to take advantage of filegroup backups and piecemeal restore. If you are using multiple filegroups for a partitioned table, you should normally assign one database file to each filegroup, and assign a distinct filegroup for each partition. Then you should spread each filegroup over the widest set of disk spindles possible for best performance. The filegroups can all reside on the same disk set: What's important is to maximize I/O performance by using as many underlying disk spindles as possible.

If you plan to use online restore with a partitioned table, you will be able to query only the partitions whose filegroups have been restored. You can query the whole table when all the filegroups for the partitions have been restored.

The SPLIT option makes an implicit assumption about the NEXT USED filegroup for a partition scheme. SPLIT requires that some filegroup be available as NEXT USED, and MERGE will free a filegroup up for use by another partition. So you need to take into account what filegroups have been freed up, and which will be used, when you use SPLIT and MERGE with multiple filegroups. Also, when you use SPLIT or MERGE, partition numbers above the event will be renumbered, making it important to keep track of which filegroup a given partition number is associated with. This adds some complexity to managing a partitioned table. For a metadata query that associates each table partition with its filegroup, see "Inspecting a Partitioned Table" in the Appendix.

You may not require multiple filegroups if you have alternate SAN-based backup and restore strategies. Then you can create the partition scheme on a single dedicated filegroup using the ALL TO clause of CREATE PARTITION SCHEME. With a single filegroup for a partition scheme and table, filegroup management becomes much simpler. For the first SPLIT, SQL Server will automatically designate the partition scheme's single filegroup as NEXT USED. However, you still need some filegroup management because for each remaining SPLIT, you must designate that single filegroup as NEXT USED. Remember, the downside of using a single filegroup is that you cannot use piecemeal restore, and with only one file in the filegroup, which is recommended for table partitioning, you will not be able to map the table to multiple LUNs on a SAN. In general, using multiple filegroups is a better method for managing a partitioned table.

Whether you use one or many filegroups for a partitioned table, you should always ensure sufficient disk space for file growth, and you should avoid autogrowth of database files by increasing the database size yourself or by creating the database files with sufficient size in the beginning. You can use SQL Server's instant file initialization to increase database file sizes very quickly.

For more information about table partitioning in relational data warehouse scenarios, see the TechNet white paper "Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server" at http://technet.microsoft.com/en-us/library/cc966457.aspx.

Partitioned Tables and Indexes

After you understand the partition function and partition scheme, you can put them to their intended use by partitioning a table. The partition function defines the boundary values and range direction, thereby establishing the initial number of partitions. The partition scheme assigns those partitions to filegroups for disk storage. Assuming you have also added files to the filegroups, you are ready to create a partitioned table.

To partition a table, you must place the table on a partition scheme. You can do this by creating the table on the partition scheme, or by creating the table's clustered index on the partition scheme. For example, the following code creates a sample partitioned table using the ON clause.

```
CREATE TABLE PartitionedTable
(PartitionColumnID int not null,
Name varchar(100) not null)
ON PS2_Right(PartitionColumnID);
```

In the ON clause, you specify the partition scheme to use and the column of the table that will serve as the partition column.

If the table already exists, there are a couple of ways to partition it without re-creating the table. If the table is clustered on the partition column, you can drop all the indexes except the clustered index, and rebuild the clustered index on the partition scheme, as in the following example.

```
CREATE CLUSTERED INDEX PT_CI ON PartitionedTable(PartitionColumnID)
WITH (DROP_EXISTING = ON)
ON (PS2_Right)
```

If the table is partitioned and clustered on the primary key, you can use ALTER TABLE.

```
ALTER TABLE PartitionedTable DROP CONSTRAINT PT_PKC
WITH (MOVE TO PS2_Right(PID))
```

Then re-create the secondary indexes so that they are aligned (see below).

In nonpartitioned tables, the ON clause of the CREATE TABLE specifies a single filegroup name, so an easy mistake is to leave out the partition column. In the current example, if the partition column is left out and just ON PS_Right is used, the error message is:

```
Msg 2726, Level 16, State 1, Line 1
Partition function 'PF2_Right' uses 1 columns which does not match with the
number of partition columns used to partition the table or index.
```

Message 2726 does not indicate that there is anything wrong with the partition function, just that no matching column from the table has been specified.

A partitioned table can be a heap, or it can have a clustered index. If you start with a heap, you can add the clustered index later, specifying the partition scheme in the ON clause of the CREATE CLUSTERED INDEX statement. If the clustered key does not contain the partition column, SQL Server will add it internally.

If you create the example table PartitionedTable without specifying a partition scheme, the table will be created on the default filegroup or other specified filegroup. Then you can partition the

25

table by creating a clustered index that identifies the partition scheme the same way that the CREATE TABLE does.

```
CREATE CLUSTERED INDEX CI_PartitionedTable ON
PartitionedTable(PartitionColumnID) ON PS2_Right(PartitionColumnID);
```

In addition, you should place a check constraint on the table to exclude partition column values that fall outside acceptable ranges, so that the partitions at the extreme left and right will not contain unusual partition column values. There is much more to say about indexes and partitioned tables, which will be covered later. For now, our focus is on partitioned tables.

The Create Partition Wizard

You can also use the Create Partition Wizard, new in SQL Server 2008, to partition a table. After you've defined the partition function and partition scheme, you can find the wizard by right-clicking the target table name in Object Explorer in SQL Server Management Studio. Click **Storage**, and then open the wizard. After you choose the partitioning column, you can use an existing partition function and partition scheme, or you can let the wizard guide you through creating them. If you use an existing partition function and scheme, the wizard will show you a summary of the partition mapping. If you want to create your own, the wizard will allow you to choose from available filegroups and set the partition function boundaries. Figure 10 shows the results of using the example partition function PF2_Right and partition scheme PS2_Right in the Create Partition Wizard.
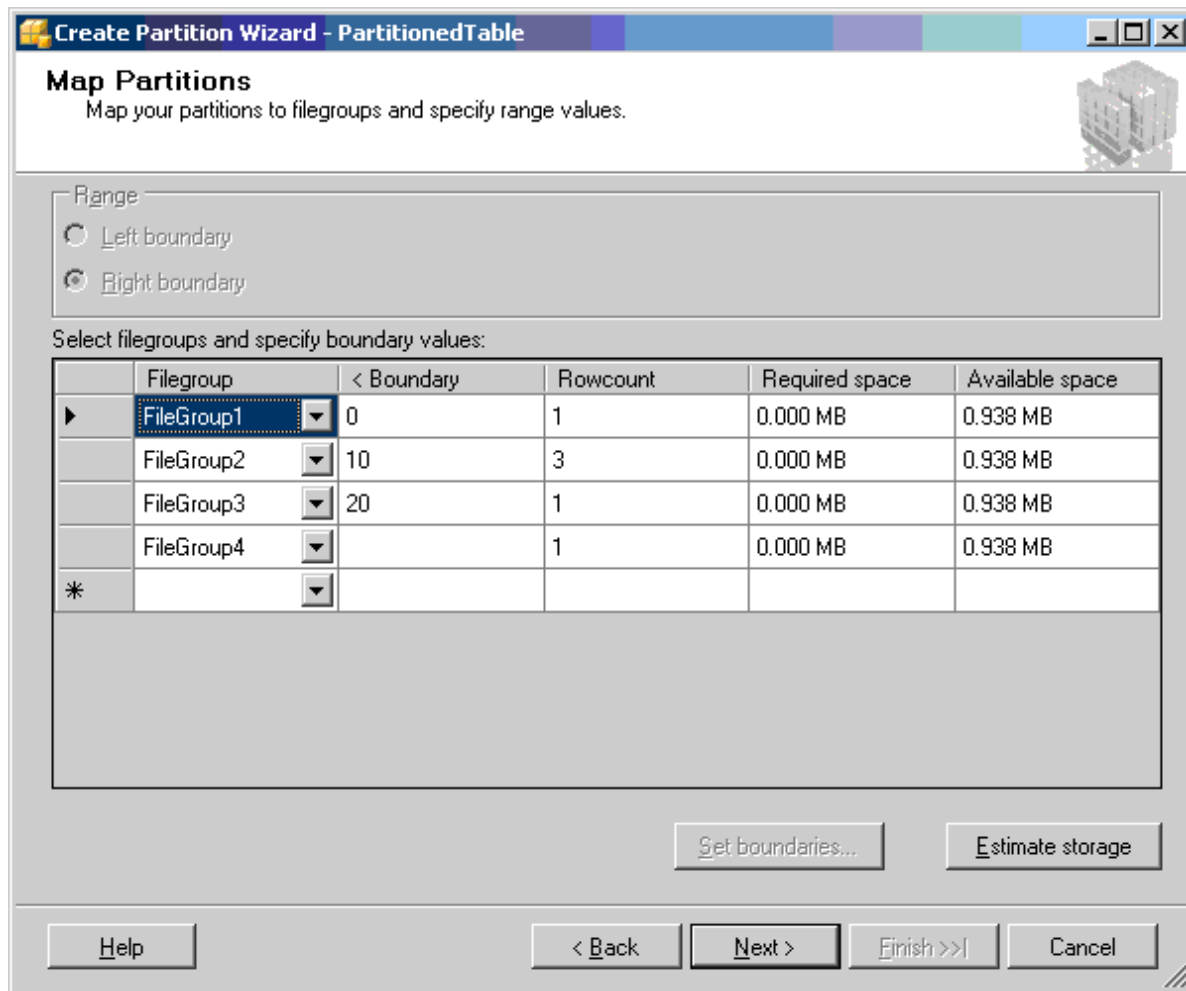
Figure 10. Mapping partitions in the Create Partition Wizard

When you click **Estimate storage**, the Create Partition Wizard inspects the table and shows the number of rows and estimated storage for each partition. If your partition column has any of the date data types, you can set their boundaries by clicking **Set boundaries**. Finally, you have the choice of partitioning the table, scheduling it, or creating a script. For more information, see "Create Partition Wizard" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/cc280408.aspx.

Indexes and Constraints on a Partitioned Table

Before considering the SWITCH operation on a partitioned table, we'll examine how indexes on a partitioned table can also be partitioned and aligned with the underlying table. It helps to consider indexes and constraints together.

There are a number of important considerations for creating indexes and constraints on a partitioned table. Table partitioning can work with heap tables without primary keys or indexes, but normally constraints and indexes are required to maintain the integrity of the table data as well as enable important partitioning benefits for management and query performance.

In a partitioned table, the partition column must be a part of:

- The clustered index key.

27

- The primary key.
- Unique index and uniqueness constraint keys.

The partition column must be part of the clustered index key. This makes sense because one of the purposes of a clustered index is to physically organize the pages of a table, and partitioning affects the physical structure of a table as well. SQL Server will internally enforce that the partition column is part of the clustered key, when the table is placed on the partition scheme and the clustered index is created.

The partition column must also be part of the primary key of the partitioned table, if one is declared, whether the primary key is clustered or nonclustered. A primary key has an underlying index to enforce uniqueness. You can place the partitioned column after the original primary key columns. For example, the **AdventureWorksDW2008** dbo.FactInternetSales table has a primary key of (SalesOrder, SalesOrderLineNumber). To partition the table on OrderDateKey, just add the partition column to the end of the new primary key: (SalesOrder, SalesOrderLineNumber, OrderDateKey).

Any unique index must also have the partition column as part of its key, so that SQL Server can enforce uniqueness across the entire set of partitions. Therefore any uniqueness constraint must also have the partition column as part of its key. If your unique index or constraint cannot contain the partitioned column, you can enforce the uniqueness using a DML trigger.

For secondary indexes that are not unique or clustered, the requirements are relaxed somewhat. Still, the benefits of including the partition column in a secondary index can be significant. When secondary indexes have the partition column as a part of their key, and use the same or equivalent partition scheme, the indexes are partitioned and are said to be *aligned* with the underlying object (heap or clustered index). SQL Server automatically adds the partition column to a secondary nonunique index as an included column if the CREATE INDEX statement does not already contain the partition column.

A secondary index does not have to use the same partition function as the underlying partitioned table to achieve index alignment, as long as the partition function used by each has the same characteristics (equivalent data types of the partition column, number and values of the boundary values, and range direction.) However, it is much more convenient to use the same partition function and partition scheme for the indexes and the underlying partitioned table.

Index alignment helps in achieving partition elimination, where the query processor can eliminate inapplicable partitions from a query plan to access just the partitions required by the query. Index alignment is also required for using the SWITCH statement, which we'll cover in the next section. If you have a nonaligned secondary index on a table and need to use the SWITCH option, you can always disable the index during the switch process and re-enable it when done.

If you use the Database Engine Tuning Advisor to recommend indexes, you have the option of choosing a partition strategy. You can have the Database Engine Tuning Advisor recommend indexes with no partitioning, with their own partitioning, or with partitioning aligned with the underlying table.

> **Note** While you can rebuild a partitioned table's index with the ONLINE option set to ON, you cannot rebuild a single partition of a partitioned index with the ONLINE set to ON. You must rebuild a single partition or a partitioned index offline because the entire partitioned table will be locked during the rebuild. You can, however, rebuild individual partitions of a partitioned index, but the entire table will be locked.

You can reorganize partitioned indexes and specify individual partitions, lists of partitions, and even ranges of partitions. Reorganizing an index is always an online operation. (For more information, see "ALTER INDEX" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms188388.aspx.)

Using the ALTER TABLE SWITCH Option

If a partitioned table uses a partition function (by referencing a partition scheme), the SPLIT and MERGE operations on the partition function cause SQL Server to redistribute relevant table data among any nonempty partitions involved. For the SPLIT operation, partitioned data and indexes are redistributed between an existing partition and a new one, and for MERGE, the data and indexes are consolidated from two partitions into the remaining one. For a partitioned table with a significant amount of data, this data movement can be costly and time-consuming.

To avoid data movement, SQL Server provides a SWITCH option on the ALTER TABLE statement. The SWITCH operation allows you manipulate the partitioned table so that SPLIT and MERGE will only operate on empty partitions, resulting in metadata-only operations. Resulting SPLIT and MERGE operations then become simply metadata operations and occur instantly. The SWITCH statement itself is a metadata-only operation, so it also happens almost immediately.

The SWITCH option is the key feature of table partitioning that makes management of large tables much more feasible. You can use the SWITCH option to take a partition's data and aligned indexes out of the partitioned table in order to manipulate the data, and then put it back in without affecting the partitioned table. You can take neighboring partitions out to combine data into one staging table, merge the two empty partitions, and then switch the data back into the partitioned table as a single partition. Perhaps most useful is that you can move the partitions forward in time by switching out the oldest partition and switching in one with fresher data. An effective strategy for aging data is the sliding window scenario. For more information, see "The Sliding Window Scenario" below.

The word SWITCH implies a symmetrical exchange, but the ALTER TABLE SWITCH option requires a TO clause that gives it a direction. You ALTER the table that has the source data, and SWITCH its partition (if it has one) TO a target table (and its partition if it has one.) In all cases, the FROM object (partition or stand-alone table) can have data, but the TO object must be empty.

SWITCH Requirements

To use the SWITCH option, at least one of the two tables involved must be partitioned using a partition scheme. It's common to refer to the stand-alone nonpartitioned table as a staging table. There are essentially three ways to use the SWITCH option:

- Switch from a partition of a partitioned table to an empty nonpartitioned staging table: In this case you alter the partitioned table.
- Switch from a staging table to an empty partition of the partitioned table: In this case you alter the staging table.
- Switch a partition from one partitioned table TO an empty partition in another partitioned table: For this option you ALTER the source partitioned table (that is, the partitioned table that has the data you want to move).

There are numerous requirements for the SWITCH command. For example, the switched tables must have the same column structure and indexes, even the same foreign key constraints. All the tables must exist and the destination table or partition must be empty. If you are switching

between two partitioned tables, the partitions must use the same partitioned column. Finally, the source and targets of the switch must exist on the same filegroup. Being on the same filegroup and in the same database, the SWITCH operation can avoid any data movement.

There are also some important requirements for indexes during a SWITCH operation:

- All indexes must be aligned.
- No foreign keys can reference the partitioned table.

Nonaligned indexes can be disabled for the duration of the SWITCH and re-enabled afterwards. There are additional requirements pertaining to indexes and partitioned tables, in particular XML and full-text indexes. For more information, see "Special Guidelines for Partitioned Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms187526.aspx.

> **Note** Whenever you switch data into or out of a partitioned table, it is important to update statistics on the table. A partitioned table keeps statistics at the table, not the partition level. If you manually update statistics on a static fact table after each new partition is loaded, you can turn off automatic statistics updating.

When you switch from a partitioned table's partition to an empty stand-alone table, the partition's data and the aligned index's partition will belong to the staging table. Only the metadata that governs which table the data and indexes belong to has changed.

Switching Data Out

To switch data out of a partitioned table:

1. Create an empty staging table on the same filegroup as the partition to be switched out with the same structure (including the partition column).
2. Create identical clustered and (optionally) nonclustered indexes on the empty staging table.
3. Optionally, create a check constraint on the stand-alone table to ensure that all values of the partition column will be in the same range as the partition that will be switched out.

The final check constraint is not required, but it is a good practice to ensure that no partition column values will be accidentally added or modified that would take them out of the original range. Plus, it allows you to switch the staging table back into the partitioned table.

Apply the ALTER TABLE command to the partitioned table, add the SWITCH option, identify the source partition, and then specify that the switch will be to the empty destination table. For example, suppose you want to switch out partition 2 on table P to an empty staging table S. The resulting SWITCH statement will be as follows.

```
ALTER TABLE dbo.P

    SWITCH PARTITION 2 TO dbo.S;
```

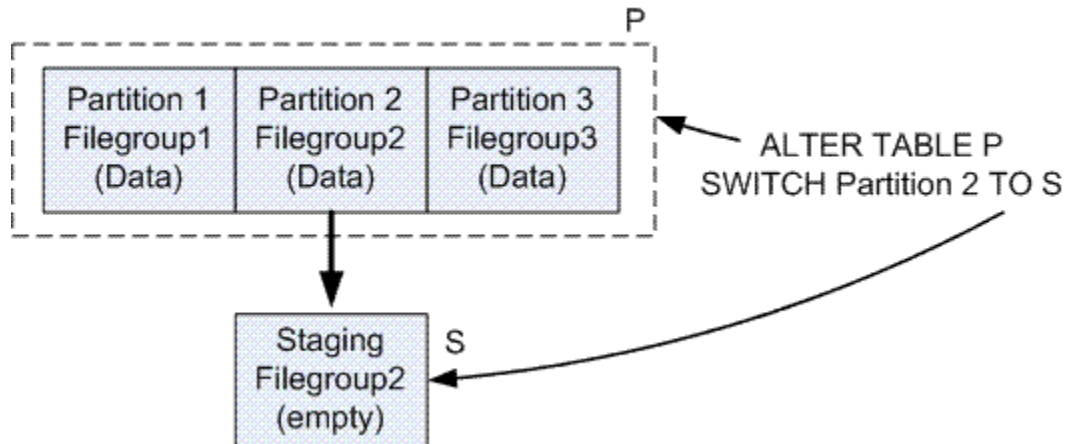Figure 11 illustrates the SWITCH operation.

Figure 11. Switching a partition out to a staging table

You can also include the partition of the nonpartitioned table (like all tables that are not linked to a partition scheme, it has only one partition, numbered 1).

```
ALTER TABLE dbo.PartitionedTable

    SWITCH PARTITION 1 TO dbo.PartitionedTableAux PARTITION 1;
```

SQL Server returns an informational message indicating that the latter PARTITION clause will be ignored. The result is that the data and indexes of the partition will be in the staging table and the partition's table and indexes will be empty.

SWITCHING Data In

When you switch from a stand-alone table into a partitioned table and exchange its data with a table's partition, you alter the stand-alone table, not the partitioned table. The stand-alone table must have the same structure, indexes, primary key, check, and foreign key constraints as the partitioned table. The table to be switched in requires one more constraint because it can only have partition column values that match the empty partition's allowable range. You accomplish this with an additional check constraint on the stand-alone table.

It is a best practice to create the indexes on the stand-alone staging table first, and then to add the required check constraint. Creating the indexes first allows SQL Server to use the indexes to validate the check constraint. If the indexes are missing, SQL Server will have to scan the table to validate the check constraint.

Also as mentioned above, the target partition in the partitioned table must be empty.

For example, the following code snippet will switch from a stand-alone table (with or without data) to an empty partition of a partitioned table.

```
ALTER TABLE dbo.PartitionedTableAux

    SWITCH TO PartitionedTable PARTITION 1;
```

Again, you can optionally add a PARTITION clause.

```
ALTER TABLE dbo.PartitionedTableAux

    SWITCH PARTITION 1 TO PartitionedTable PARTITION 1;
```

The latter command results in an informational message saying that the partition clause will be ignored because the table being altered is not a partitioned table. Figure 12 illustrates switching in.
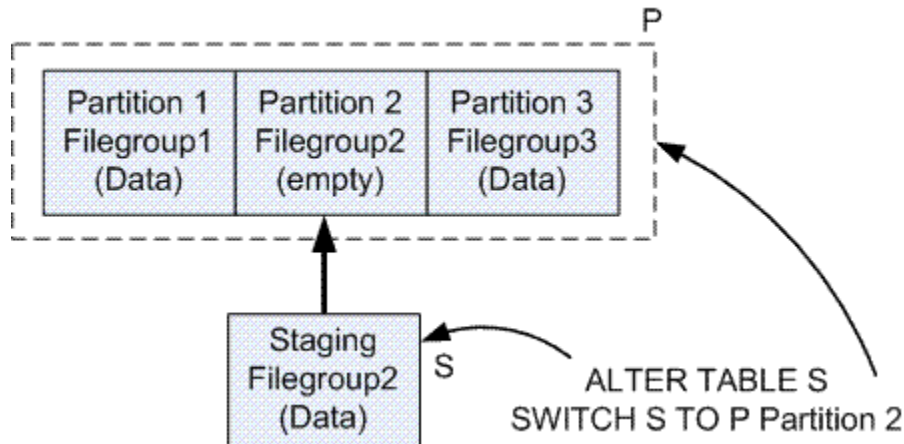
31

Figure 12. Switching from a staging table into a partition

SWITCHING Between Two Partitioned Tables

If you are using the SWITCH statement to exchange partitions between two partitioned tables, the additional PARTITION option in the TO clause is required. The key thing to note is that when switching out, you alter the source partitioned table, specify its partition right after the SWITCH clause, and then specify the target partitioned table in the TO clause. For example, the Sliding Window sample code for SQL Server 2008, Sliding.sql, includes the following command to switch a partition from the TransactionHistory table to an empty partition in the TransactionHistoryArchive table.

```
ALTER TABLE [Production].[TransactionHistory]

SWITCH PARTITION 1

TO [Production].[TransactionHistoryArchive] PARTITION 2;
```

In this example, the keyword PARTITION is required in both places because the data is being switched from one partitioned table to another.

When SWITCH is used to exchange data between two partitioned tables, SQL Server verifies internally that all the rows from the source table's partition will fall correctly into the specified target table's partition.

In general, when a switch occurs, both the partition's data and its partitioned (aligned) indexes are switched out or in automatically. No additional work is required for the indexes: The data and indexes participate in the SWITCH operation together–provided that the indexes are truly aligned.

If you have any nonaligned indexes on the partitioned table, you can disable them during the switch process and re-enable them afterwards. However, this will cause the nonaligned secondary indexes to be rebuilt, which is not a metadata-only operation.

Using the Manage Partition Wizard to Perform SWITCH Operations

You can use the Manage Partition Wizard to generate the script necessary to switch a partition out of a table to an unpartitioned staging table. Just right-click the table name in Object Explorer in SQL Server Management Studio, and then under Storage, click **Manage Partition**. On the Select a Partition Action page, click **Create a staging table**. The wizard will suggest a name for

the staging table to switch the partition out, or you can choose your own. You can then choose a partition of the designated table, and whether to generate a script, run immediately, or schedule the switch. It's best to view the script first to make sure it will do what you need. The resulting script will contain commands to create:

- The staging table.
- The necessary clustered and nonclustered indexes.
- Foreign keys.
- A check constraint to preserve the partition column's range.

The script does not contain the ALTER TABLE command to actually make the switch.

Partition-Aligned Indexed Views

SQL Server 2008 introduces *partition-aligned* indexed views. These are indexed views that can participate in the ALTER TABLE SWITCH operations and therefore also in a sliding window. The materialized aggregate data of the indexed view can be switched in and out of a partitioned table to another partitioned table, without the need to rebuild the aggregate values.

SQL Server 2005 already enabled creating indexed views on a partitioned table. If you created the indexed view's clustered index on the same or similar partition scheme as the underlying table, the result is a partitioned indexed view. However, in SQL Server 2005, you could not switch any partition in or out of the partitioned table while a partitioned index was defined on it. As a result, aggregates and other materialized data in the partitioned view had to be dropped and re-created each time a switch occurred.

With an indexed view in place, queries accessing a table may be able to take advantage of query rewrite, that is, a transparent transformation by the query processor, to use the indexed view instead of the underlying tables it is defined on, even though the query might not directly reference the indexed view. In the case of large tables, it is common for an indexed view to materialize aggregate values across large numbers of rows. If those aggregates remain available in the indexed view, they can greatly improve performance of queries that both do and do not explicitly reference the indexed view. Because it can take a long time to re-create the indexed view, being able to switch partitions in and out of an indexed view instead of having to re-create the index can save a great amount of time.

In SQL Server 2008, if you follow the rules for creating a truly partition-aligned indexed view, you can use SWITCH, MERGE, and SPLIT operations and the partitioned table without having to drop and then re-create the indexed view. As a result, the indexed view remains online and available to the application. You can do this because SQL Server can switch the materialized data in a partitioned indexed view and switch it in and out of the portioned index of a partitioned table.

There are a number of requirements for an indexed view to be partition-aligned with a partitioned table:

- The partitioned table and indexed view must share the same or equivalent partition functions (same number of and values of boundary values, same data type, and so on).
- The indexed view must contain the partition column in its SELECT list.
- Any GROUP BY statement must also include the partition column of the underlying table.
- An indexed view can only be partition-aligned with one partitioned table at a time.

33

You can find details about the requirements for implementing a partition-aligned indexed view, along with sample code for your own testing, in "Partition Switching When Indexed Views Are Defined" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/bb964715.aspx. The sample code contained in that entry switches a table's partition out to another partitioned table, along with the corresponding partition of the indexed view.

Suppose you want to switch a partition out from a partitioned table that has a partition-aligned indexed view defined, and you do not want to rebuild the indexed view. You can execute the SWITCH directly against the partitioned table, and then you can execute it indirectly against its partitioned-aligned indexed view, as shown in Figure 13.
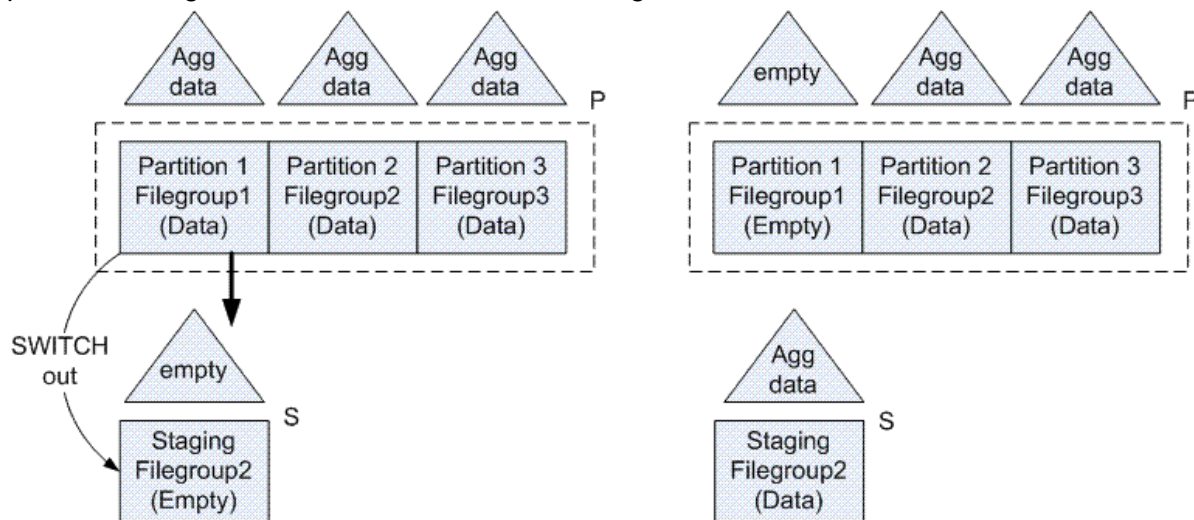
Figure 13. Before and after scenarios for switching out a partition with a partition-aligned indexed view

In the scenario on the left, we begin by building the usual staging table S for switching a partition out of partitioned table P. We build S on the same filegroup as the target partition.

At this point we have an option: We could switch out the partition and just move the data to S. There is no requirement, when switching out from a partitioned table with a partition-aligned indexed view, for the staging table to also have an indexed view. If we go down this path, SQL Server will empty out the aggregate data from partition 1's corresponding partition in the indexed view.

However, it's more efficient to build an indexed view on staging table S using the same view definition and clustered index as the partitioned table. Then when switching the table out, we can preserve the aggregate data along with the data in S, as shown on the right side of Figure 13. This way, SQL Server does not have to delete any of the data in the aggregate; if we truncate the staging table S, removal of the aggregate data will not affect the partitioned table.

To switch data back in, an indexed view must be created on the staging table that matches the indexed view on the partitioned table. Figure 14 illustrates the process of switching in.
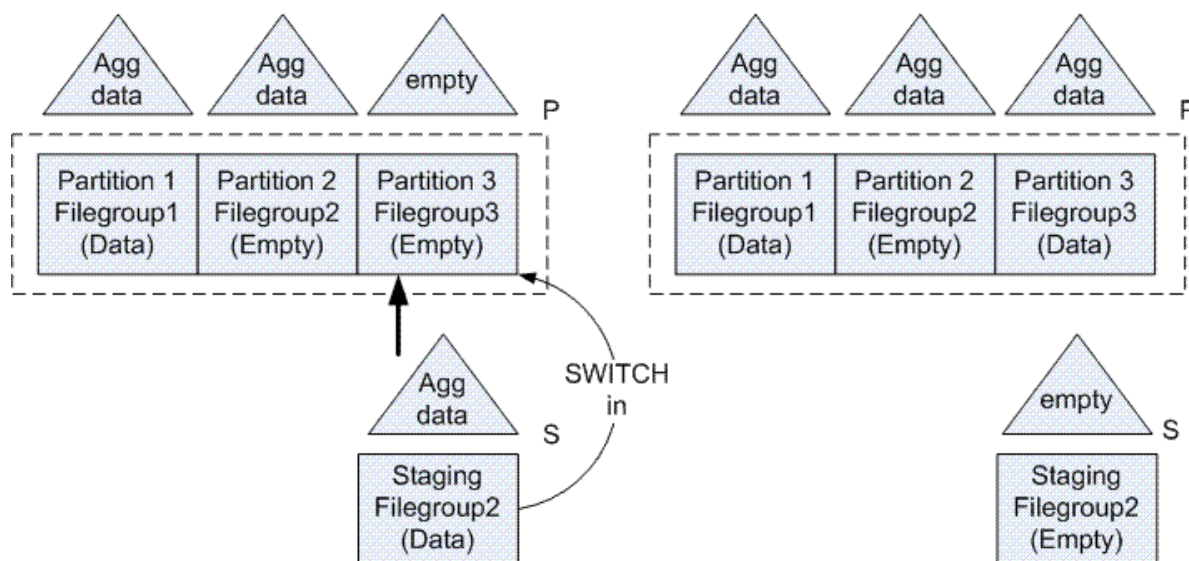
Figure 14. Before and after scenarios for switching in a partition with a partition-aligned indexed view

To switch data into a partitioned table with a partition-aligned indexed view, you must have a similar and compatible indexed view declared on the staging table, which is shown on the left side of Figure 14. We're switching into partition 3, which is empty, and so is the partition of the indexed view. SQL Server performs the necessary metadata operations to place the staging table's rows, indexes, and indexed view into the appropriate partition of the partitioned table, and in return the staging table ends up as a table, indexes, and an indexed view with no rows.

Looking at Figures 13 and 14, you can see that it is now possible in SQL Server 2008 to maintain a sliding window scenario with indexed views declared on the underlying table. The switch occurs instantly, and you do not have to rematerialize the data of the indexed view during each periodic switch. You can then archive the aggregate data as well as the table's row data if you choose.

Partition Elimination and Query Performance

SQL Server 2008 adds increased support for analyzing query plans on partitioned tables in several ways:

- A new partition-aware seek operation to determine partition elimination
- Two levels of the partition-aware seek, called a skip scan
- A Partitioned attribute for physical and logical operators that appears in the query plan
- Partition summary information, including the partition count and the partitions accessed

These operations can all be detected using the SQL Server 2008 graphical actual query plan in SQL Server Management Studio, SET SHOWPLAN_XML, or SET STATISTICS XML.

For limitations on other ways of displaying query plan, see "Displaying Partition Information by Using Other Showplan Methods" in "Query Processing Enhancements on Partitioned Tables and Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms345599.aspx. For more information about table partitioning and query performance

in SQL Server 2008, see "Data Warehouse Query Performance" at
http://technet.microsoft.com/en-us/magazine/2008.04.dwperformance.aspx.

Partition-Aware Seek Operation

A key benefit of table partitioning is partition elimination, whereby the query processor can eliminate inapplicable partitions of a table or index from a query plan. The more fine-grained the query filter is on the partition column, the more partitions can be eliminated and the more efficient the query can become. Partition elimination can occur with table partitions, index-aligned partitions, and partition-aligned indexed views in both SQL Server 2008 and SQL Server 2005. Partition elimination can also occur with nonaligned indexes.

SQL Server 2008 adds a new partition-aware seek operation as the mechanism for partition elimination. It is based on a hidden computed column created internally to represent the ID of a table or index partition for a specific row. In effect, a new computed column, a partition ID, is added to the beginning of the clustered index of the partitioned table.

Finding the partition-aware seek operation helps to determine whether a partition elimination has occurred. In a query plan, it shows up as Seek Keys[1] with the level number in brackets and the table's partition ID indicated afterwards. In addition, there are additional features in SQL Server 2008 query plans that can contain partition information.

For example, let's inspect the actual query plan for the following query on the **AdventureWorksDW2008** FactInternetSales database.

```
SELECT *

FROM FactInternetSales

WHERE OrderDateKey BETWEEN 20030402 AND 20030822

AND SalesOrderNumber = 'SO51922'
```

In this case, SQL Server uses the nonclustered index built by the primary key to resolve the query. You can view the new partition-aware seek operation in the ToolTip when you mouse over an operator, or in the **Seek Predicates** node of the operator's properties. Figure 15 shows the **Seek Predicates** portion of the ToolTip for the nonclustered index seek on the FactInternetSales table.
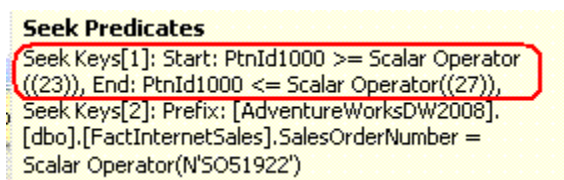

Figure 15. The Partition-Aware Seek property in the ToolTip view of a graphical query plan

In addition, there is summary information about the partition access for this operator at the bottom of the ToolTip, as shown in Figure 16.

Figure 16. The Partitioned property and Actual Partition Count values in the ToolTip view of a graphical query plan

Figure 16 illustrates that if the index seek is performed on a partitioned table, there is a new **Partitioned** property that appears with the value True. In addition, the number of partitions accessed in the query is listed in Actual Partition Count.

You can also find out which partitions were actually accessed in the **Properties** dialog box for the same operator, as shown in Figure 17.

Figure 17. Determining the number of partitions accessed in the operator's **Properties** dialog box

In this case you can see that partitions 23 through 27 were accessed, on the Actual Partitions Accessed line.

With this information, you can determine that the query successfully used partition elimination by accessing only the relevant partitions as listed in the query filter. SQL Server was able to eliminate other partitions of the table from the query, thereby reducing the amount of table access required.

Partition elimination is most useful when a large set of table partitions are eliminated, because partition elimination helps reduce the amount of work needed to satisfy the query. So the key point is to filter on as small a number of partitions as possible based on filtering the partition column.

Skip-Scan: Seek Keys

You may have noticed that in Figure 17 there was also a Seek Keys[2] predicate: Figure 18 outlines it more sharply.
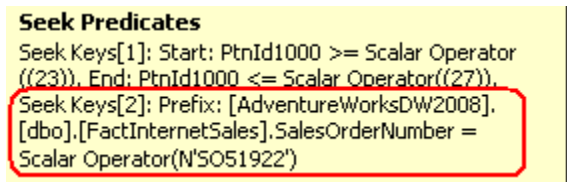

Figure 18. Detecting the Seek Keys[2] skip scan

This is the "skip-scan" operation, which is a second level of seek operation after the first level called out as Seek Keys[1]. When both levels are present, SQL Server can use the first-level (Seek Keys[1]) to find all the partitions required to satisfy the query, which you can see in the two scalar operators, (23) and (27), that defined the partition range required by the filter of the query. The second-level seek then accesses the combination of conditions that can be satisfied by the index, in this case SalesOrderNumber = 'SO51922', for each partition. Whenever you see two levels of Seek Keys[], a skip scan has occurred.

Join Collocation

If you query two tables that are partitioned with compatible partition functions, you may be able to join the two tables on their partition column. In that case, SQL Server may be able to pair up the partitions from each table and join them at the partition level. This is called 'join collocation', implying compatibility between the partitions of the two tables that is leveraged in the query.

For further explanation and an example of join collocation, see the "Interpreting Execution Plans for Collocated Joins" subtopic of "Query Processing Enhancements on Partitioned Tables and Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms345599.aspx. You can use the code sample at the end of the article to observe join collocation. The script generates a table called fact_sales. Just duplicate the code and generate a similar fact table called fact_sales2 and partition it using the same partition function, and create its clustered index using the same partition scheme as fact_sales1. When you join the two tables, as in the following sample query, join collocation will occur.

```
SELECT TOP 10 F1.date_id,
      SUM(F2.quantity*F2.unit_price) AS total_price
FROM fact_sales AS F1
JOIN fact_sales2 AS F2
      ON F1.date_id = F2.date_id
      AND F1.product_id = F2.product_id
      AND F1.store_id = F2.store_id
WHERE F1.date_id BETWEEN 20080830 AND 20080902
GROUP BY F1.date_id ;
```

Two partitions will be touched, but only a small part of each, so the query returns values quickly. Figure 19 shows the resulting query plan.
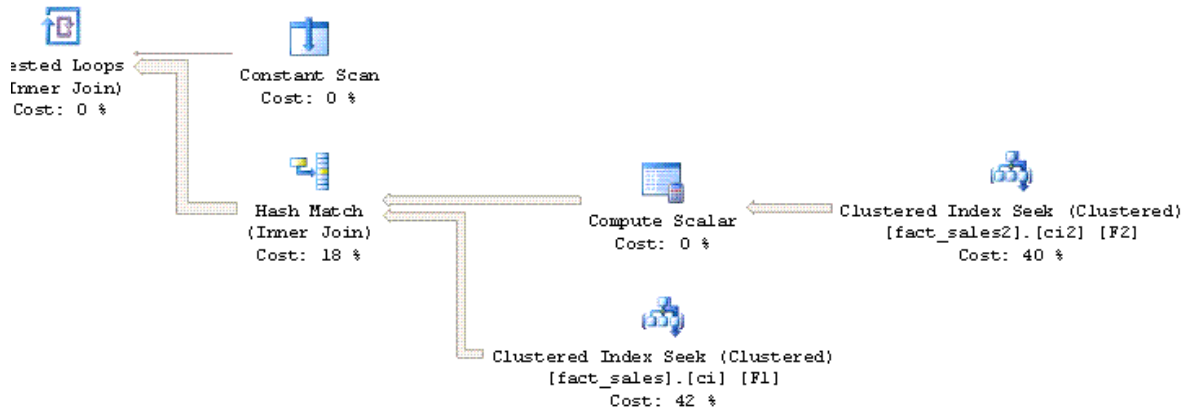
Figure 19. Join collocation indicated by the Constant Scan operator

The Constant Scan operator returns rows for partitions 2 and 3. The Nested Loops join reads the joined partitions based on the partition numbers to assemble the results. For an example of a parallelized plan, see the same subtopic in SQL Server 2008 Books Online.

Join collocation is a logical, not a physical phenomenon: It is not necessary for the partitions of the joined tables to reside on the same filegroups, just that they have compatible partition boundaries from their partition functions. SQL Server is able to detect that the join operation can be pushed down to the partition level, and that is what causes the collocation.

Partitioned Table Parallelism

Parallel operations are a key benefit of table partitioning. Indexes can be rebuilt in parallel, and the query processor can take advantage of multiple partitions to access a table in parallel.

If a partitioned table is sufficiently large and at least two CPU cores are available to SQL Server, a parallel execution strategy is used across the partitions that the query filter resolves to. Generally speaking, SQL Server attempts to balance the number of threads assigned to various partitions. The **max degree of parallelism** setting (which you set by using the **sp_configure** stored procedure or the MAXDOP query hint) determines the available thread count.

However, if the filter of the query specifically calls out ranges that determine a subset of the partitions so that partition elimination can occur, the number of partitions accessed will accordingly be less.

SQL Server 2005 was optimized for queries filtered to one partition. For such queries, more than one available thread could be assigned to scan the partition in parallel. However, for a filtered query touching more than one partition, only one thread could be assigned to any given partition. In SQL Server 2008, if the number of available threads is greater than the partitions accessed by a filtered query, more than one thread can be assigned to access data in each partition. This can improve performance in filtered queries that access more than one partition. An explanation of the assignment of parallel threads to partitions is presented in the "Parallel Query Execution Strategy for Partitioned Objects" subtopic of the "Query Processing Enhancements on Partitioned Tables and Indexes" topic in SQL Server 2008 Books Online at http://technet.microsoft.com/en-us/library/ms345599.aspx. (See also "Data Warehouse Query Performance" at http://technet.microsoft.com/en-us/magazine/cc434693.aspx.)

Bitmap Filtering

SQL Server 2008 can enhance parallelized query plans on partitioned tables through use of a bitmap filtering. Bitmap filters were introduced in SQL Server 2005 and improved in SQL Server 2008: In SQL Server 2008, bitmap filters can be introduced dynamically by the query optimizer during query plan generation. This is particularly relevant for data warehouses that join large fact tables to dimension tables in a star schema. A table that is partitioned on an integer that simulates a date can take advantage of optimized bitmap filtering. For more information about bitmap filtering, see "Optimizing Data Warehouse Query Performance Through Bitmap Filtering" in SQL Server 2008 Books Online [at http://msdn.microsoft.com/en-us/library/bb522541.aspx](http://msdn.microsoft.com/en-us/library/bb522541.aspx).

Data Compression

Data compression provides a significant potential for query performance improvement for large tables, including partitioned tables. By reducing the storage space for tables and indexes, the number of I/Os are reduced. In the case of large partitioned tables, a reduction of I/Os, especially physical I/Os, is very helpful. Data compression does introduce additional CPU activity, so it is important to make sure that the gain in I/O offsets any additional CPU load. (Like table partitioning, data compression is available only in SQL Server 2008 Enterprise.)

You can specify data compression for an entire partitioned table, index, or indexed view, and you can specify compression at the table or index's partition level. You can compress partitions one at a time using the ALTER TABLE REBUILD and ALTER INDEX REBUILD commands. To compress an indexed view, compress the indexed view's clustered index. The syntax for rebuilding (compressing or decompressing) more than one partition, but not all the partitions, is worth noting. You must add the PARTITION = ALL clause, as in the following sample code, which rebuilds partitions 23 and 27.

```
ALTER TABLE dbo.FactInternetSales

REBUILD PARTITION = ALL

WITH (DATA_COMPRESSION = PAGE ON PARTITIONS (23,27) )
```

The PARTITION = ALL clause is required, even though not all partitions are actually being compressed. Because this will rebuild all the partitions, it is a best practice to compress only one partition at a time if you do not need to compress all the partitions.

You can also use the Manage Compression Wizard (found in Object Explorer in SQL Server Management Studio: Just right-click a table or index, and then click **Storage**). In addition, you can use the **sp_estimate_data_compression_savings** system stored procedure to get an estimate of the potential space savings. You can even use the $PARTITION function with it, as in the following example.

```
EXEC sp_estimate_data_compression_savings

    @schema_name = 'dbo'

    , @object_name = 'FactInternetSales'

    , @index_id = NULL

    , @partition_number = $PARTITION.pfn_OrderDateKeyRight(20020301)

    , @data_compression ='PAGE' ;
```

Data compression can be very beneficial, especially for tables, but before you attempt to compress any indexes, make sure that they are not fragmented. You may be able to achieve the desired compression at the index level just by reorganizing the index.

40

When you use SPLIT and MERGE on a partitioned table that is compressed, the resulting new or consolidated partitions will inherit the compression setting of the original or partner partition. When you switch a partition in or out, the compression settings between the staging table and the partition must match. For more details, see "How Compression Affects Partitioned Tables and Indexes" under "Creating Compressed Tables and Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/cc280449.aspx.

Additional Partitioning Support in SQL Server 2008

Many new features of SQL Server 2008 have uses or implications for table partitioning:

1. **Controlling lock escalation granularity within a partitioned table.** In SQL Server 2008, you can control lock escalation at the partition level. By default, if a table is partitioned, setting the lock escalation value to AUTO causes locks to escalate to the partition level only, not the table level. This can reduce lock contention but it increases the number of locks taken.

2. **Partitioning a Table with a FILESTREAM Column.** A table with a FILESTREAM column must have a unique constraint on a ROWGUID column, declared either as a constraint or as a unique secondary index. If the natural partitioning column is not a ROWGUID, it may be possible to create the ROWGUID value as a function of the natural partitioning column, such as a date, and then specify corresponding range values for the ROWGUID that match the desired date ranges. If you do not partition a table on the FILESTREAM's required ROWGUID column, you can maintain the uniqueness of the column based on an index and then disable the index (and thereby disable FILESTREAM operations) during any SWITCH operations. When the SWITCH is completed, you can enable (rebuild) the unique index, and when that is done, FILESTREAM operations can continue.

3. **Filtered indexes.** In SQL Server 2008, you can place filters (limited WHERE clauses) on secondary indexes. Filtered indexes are fully compatible with table partitioning. Only secondary nonclustered indexes can be filtered. Just as with ordinary indexes, in order to enable the SWITCH operation you must either include the partition column as part of the index key, or disable the index when executing SWITCH operations. The filtered index must be aligned with the partitioned table.

4. **Change Tracking.** Change tracking is used for maintaining synchronization between application data and SQL Server tables. You can apply change tracking to a partitioned table. However, the SWITCH command will fail if either table involved in the SWITCH has change tracking enabled. For more information, see "Configuring and Managing Change Tracking" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/bb964713.aspx.

5. **Change Data Capture.** The Change Data Capture utility captures all INSERT, UPDATE, and DELETE activity against a partitioned or a nonpartitioned table. But if you want to use the SWITCH operation on the table, you must explicitly state so in the sys.sp_cdc_enable_table command. If you do not, any SWITCH will be prevented. For more information, see "sys.sp_cdc_enable_table" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/bb522475.aspx.

6. **Sparse columns.** Sparse columns provide another way of more efficiently storing database data, by storing columns that are mostly NULL in an alternate location. However, A sparse column cannot be a part of a clustered index (or primary key); it also

cannot be the partitioning key of a partitioned table, whether clustered or a heap. However, you can partition a secondary index on a sparse column. This could be useful if you want to partition just the secondary index.

7. **Full-text Search**. Although you can have full-text indexes on a partitioned table, the SWITCH command will be prohibited. The full-text index is treated like a nonaligned index.

8. **Replication.** Table partitioning is compatible with transactional replication, and even switching partitions in and out can be replicated, provided the partitioned tables are set up correctly on the Publisher and Subscriber tables, and the available partitioning options in replication are used. Merge replication does not support partition switching. For more information, see "Replicating Partitioned Tables and Indexes" in SQL Server 2008 Books Online at <u>http://msdn.microsoft.com/en-us/library/cc280940.aspx</u>.

The Sliding Window Scenario

The sliding window scenario is a method for aging old data out and bringing new data into a partitioned table on a periodic basis. Using a time-based or date-based partitioned table, the technique employs the SWITCH option of ALTER TABLE, along with the MERGE and SPLIT operations of ALTER PARTITION FUNCTION, to move the table's partitions forward in time one partition at a time, leaving the overall number of partitions the same. (For more information, see "Designing Partitions to Manage Subsets of Data" in SQL Server 2008 Books Online at <u>http://msdn.microsoft.com/en-us/library/ms191174.aspx</u>.)

The key feature of a sliding window scenario is that the partitions represent time linearly, and the window of time slides forward period by period, keeping the overall number of partitions in the partitioned table constant. The window moves forward partition by partition, reflecting the period of time chosen by the boundary values of the partition function.

Sliding Window Steps

In general, a sliding window consists of the following steps:

1. Create a nonpartitioned archive table with the same structure, and a matching clustered index (if required). Optionally you can add matching secondary indexes and constraints. Place it on the same filegroup as the oldest partition.

2. Use SWITCH to move the oldest partition from the partitioned table to the archive table.

3. Remove the boundary value of the oldest partition using MERGE.

4. Designate the NEXT USED filegroup.

5. Create a new boundary value for the new partition using SPLIT (the best practice is to split an empty partition at the leading end of the table into two empty partitions to minimize data movement.).

6. Create a staging table that has the same structure as the partitioned table on the target filegroup.

7. Populate the staging table.

8. Add indexes.

9. Add a check constraint that matches the constraint of the new partition.

10. Ensure that all indexes are aligned.

11. Switch the newest data into the partitioned table (the staging table is now empty).

12. Update statistics on the partitioned table.

In addition, to protect the partitioned table from allowing any data values from beyond the desired boundary range, a check constraint over the entire boundary range can be applied.

There are a number of things you can do to make the sliding window scenario as efficient as possible:

- Minimize data movement by using SPLIT and MERGE on empty partitions. Switching out the oldest data partition and leaving an empty partition in its place allows you to merge an empty partition. The same is true when you use the SPLIT operation to create the partition for the newest data: It is initially an empty partition, so the SPLIT operation does not cause any data movement.
- Create the staging table for new data as initially a heap, load the data, and then build indexes (and indexed views, if any) as a preparation for switching.
- Update statistics on the table after every partition switching cycle (because statistics are kept at the table level, this operation can be time-consuming and resource intensive).

In a sliding window scenario, the oldest data switched out of the partition can be archived or deleted, depending on business requirements. We'll deal with archiving scenarios in the next section.

A Sliding Window Illustration

To illustrate a sliding window, we will consider the FactInternetSales table of **AdventureWorksDW2008**. The order date for each sale fact is recorded in OrderDateKey, an integer column representing order dates as YYYYMMDD. Suppose you partition the table based on month. The oldest order has an OrderDateKey of 2000701, and the newest data is 20040731. A likely requirement is to age the data month by month, based on the order date, which is recorded in the OrderDateKey.

To minimize data movement during the load and archive steps, an empty partition is kept on each end of the partitioned table. That helps ensure that only empty partitions are ever split or merged. The sliding window procedures are then the same whether RANGE LEFT or RANGE RIGHT is used in the partition function.

For this illustration, the partition function is created using RANGE RIGHT. The boundary values specify the beginning of the month because RANGE RIGHT boundary values are the lowest boundary of a partition's range, ranging from 20010701, 20010801, 20010901, and so on, through 20040801. The table initially only contains data from 20010701 through 20040731, so specifying the final boundary value of 20040801 ensures that the last partition will be empty.

The partitioned table also contains a check constraint to ensure that only values from 20010701 through 20040731 are allowed in the table.

Our first step is to archive the oldest month's data by switching the data from partition 2 into a nonpartitioned archive table. The archive table is placed on the same filegroup as partition 2. The result of the switch is shown in Figure 20.
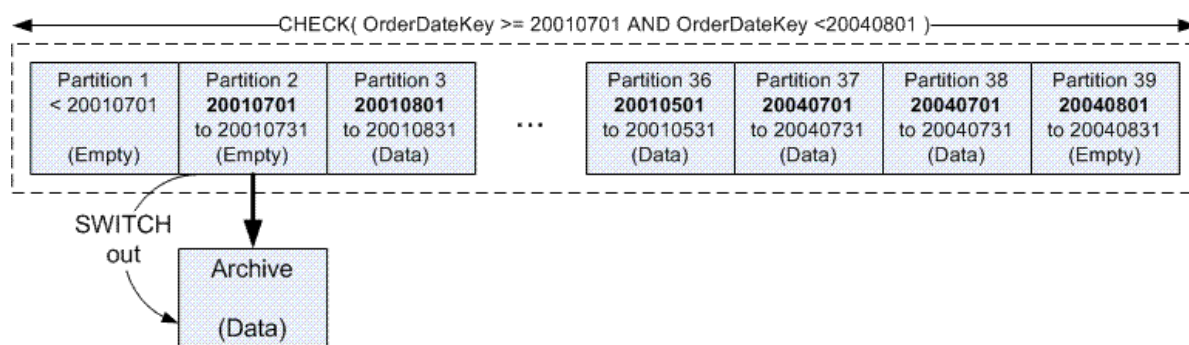
43

Figure 20. Switching out the oldest partition containing data

After the switch has occurred, the boundary point 20010701 is removed using MERGE. The table's check constraint is revised ensure that no OrderDateKey values earlier than 20010801 can enter the table, ensuring that the first open-ended partition remains empty. The result is shown in Figure 21.
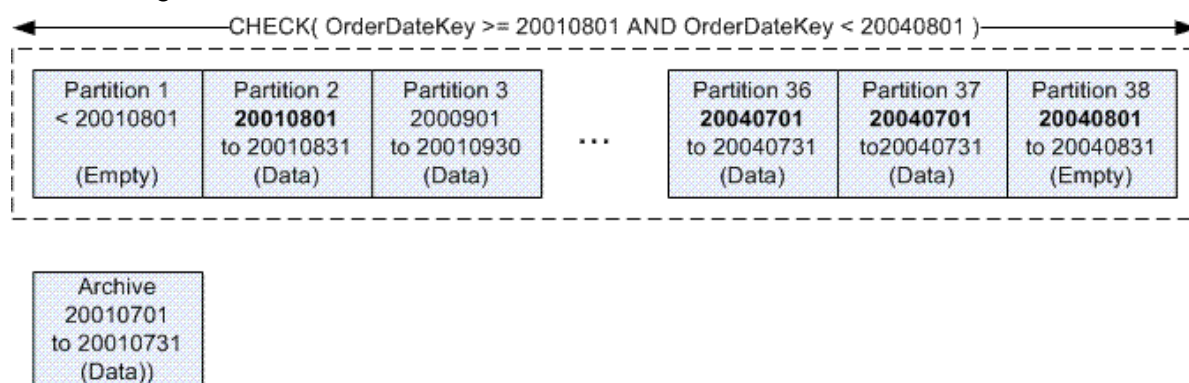

Figure 21. Result of removing the oldest data partition using MERGE

Note that the MERGE operation, by removing one partition, caused all the partitions to be renumbered. At this point data from the archive table can be copied out to an archive location, and then the archive table can be either truncated for later use, or dropped.

The next step is to load new data into the partitioned table from a staging table into partition 38. The staging table is loaded on that filegroup. To prepare for the SPLIT operation, a new filegroup is designated as NEXT USED. A new boundary value of 20040901 is added to the partition function using SPLIT, creating a new partition 39, leaving the table momentarily with two empty partitions at the leading end. The check constraint on FactInternetSales is revised to ensure that no values greater than 20040831 are allowed into the table, ensuring that the new partition 39 remains empty. The result is shown in Figure 22.
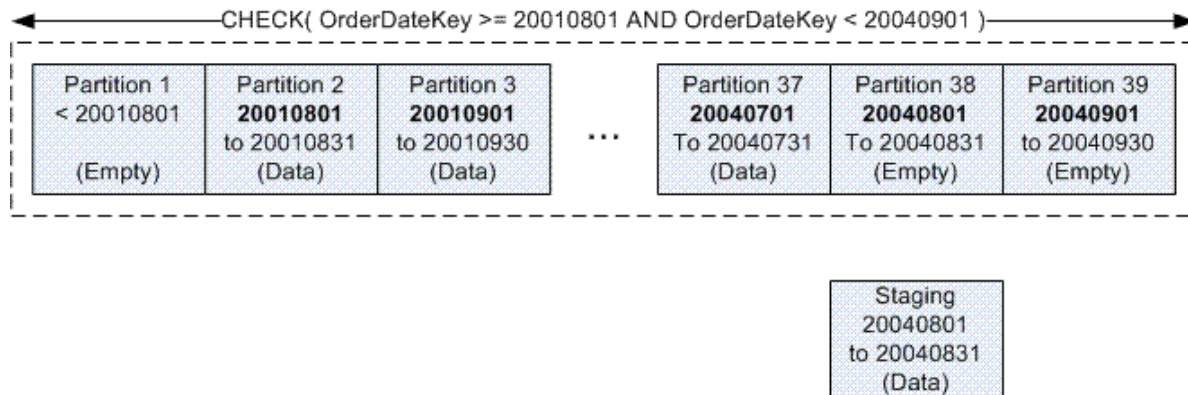
Figure 22. Adding a new partition by splitting the empty partition at the leading end of the partitioned table

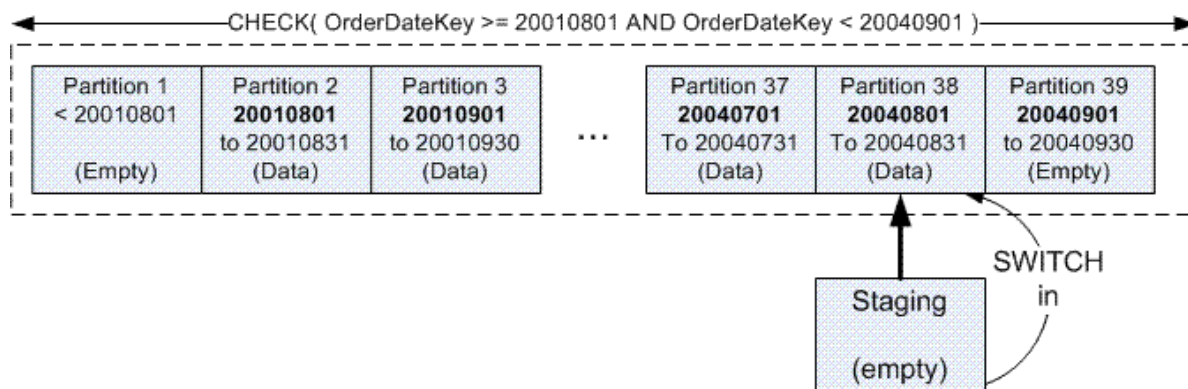Now the SWITCH operation is performed, as shown in Figure 23.

Figure 23. A newest month's data is switched into the table

At this point the staging table is empty and can be dropped.

By maintaining an empty partition at both ends of the table, only empty partitions are either merged or split, eliminating any data movement between partitions. Also empty partitions on each end of the partitioned table makes the basic operations the same whether the partition function uses RANGE RIGHT or RANGE LEFT.

Manipulating a sliding window can be complex, mostly because you must manage partition numbers, filegroups for moving the data forward, and all the constraints on the partitioned and staging tables to accomplish the SWITCH, MERGE, and SPLIT operations. There are, however, some tools available to assist with managing the partitions in a sliding window.

The Manage Partition Wizard

You can use the Manage Partition Wizard to automatically switch the oldest data out to an archive table and newest into the partitioned table from a staging table, for a RANGE LEFT partitioned table. Just right-click the partitioned table, click **Storage**, open the wizard, and then click **Manage partitioned data in a sliding window scenario**. You can switch in, switch out, or both. If you click **Switch Out**, you enter a new boundary value for the next newest partition. The

wizard will generate the switch out (archiving) table, but relies on you to prepare the switch in (staging) table.

You can have the code executed immediately, on schedule, or else just generate the code to some location.

Sliding Window Between Two Partitioned Tables

SQL Server 2008 Books Online has an interesting variation on the sliding window using the **AdventureWorks** database. The Sliding.sql script provided in the Samples for SQL Server 2008 on CodePlex archives the oldest partition (September 2003) of the partitioned Production.TransactionHistory table to a new partition in the Production.TransactionHistoryArchive table. This is a good illustration of using SWITCH to move data from one partitioned table to another. (See also "Designing Partitions to Manage Subsets of Data" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms191174.aspx.)

Two Tools to Help Automate a Sliding Window

Automating a sliding window can be very challenging. There are a couple of unsupported approaches provided by Microsoft.

A general-purpose utility called the Partition Management Utility for SQL Server 2008 can be downloaded from CodePlex at http://www.codeplex.com/SQLPartitionMgmt. The Partition Management Utility (ManagePartition.exe) contains five basic operations. Although developed by the Microsoft SQL Server CAT team, this tool is not officially supported by Microsoft. The utility contains the following commands:

- ClearPartition clears a single partition from a table. This will switch a partition out to a staging table on the same filegroup, with a name of your choice, and optionally drop the staging table after the switch.
- CreateStagingFull creates an empty staging table with all necessary indexes, foreign keys, and check constraints matching a given partition.
- CreateStagingNoindex creates a staging table for a given partition with no indexes for fast loading.
- CreateStagingClustedIndex creates a staging table with only a clustered index.
- IndexStaging creates the indexes on a specified staging table that has been created using CreateStagingNoindex and loaded with data.

To automate a sliding window scenario with the Partition Management Utility, you could take the following steps:

1. Use ClearPartition on the oldest data partition, keeping a staging (archive) table with the name you supply.
2. Remove the boundary value of the oldest partition using MERGE.
3. Designate the NEXT USED filegroup.
4. Create a new boundary value for the new partition using SPLIT.
5. Use CreateStagingNoindex to create an empty staging table for data load on the NEXT USED filegroup.
6. Populate the staging table with new data.
7. Use IndexStaging to create the indexes and constraints on the staging table.
8. Switch the staging (newest data) table into the partitioned table.

9. Update statistics on the partitioned table.

The Partition Management Utility handles the task of determining the correct filegroup based on the specified partition number, and it also adds the appropriate constraints. You must be aware of your partitions and available filegroups.

A second less general-purpose tool uses Transact-SQL to automate the SPLIT and MERGE operations. The Transact-SQL code is available in the MSDN® article, "How to Implement an Automatic Sliding Window in a Partitioned Table on SQL Server 2005" at

http://msdn2.microsoft.com/en-us/library/aa964122.aspx. Although targeted for SQL Server 2005, the code operates with SQL Server 2008 as well. The article illustrates its technique using a sample partitioned table using RANGE RIGHT and daily data for 60 days, based on a datetime column. Two partitioned tables are created: a main partitioned table containing current data, and an auxiliary table containing archived data. To create the partition for a new day's worth of data at the end of the main partitioned table, they provide a stored procedure that automatically retrieves the next boundary ID, sets the NEXT USED filegroup, and then uses SPLIT to create the new partition. To archive the oldest partition, the article supplies a stored procedure that retrieves the boundary value for partition 2 of the table, adds a new partition to the auxiliary table, and then switches that data into the auxiliary table. However, you will have to add your own filegroup management and archiving techniques.

Other Scenarios

The Rotating Window Scenario

Instead of a window that moves forward in time by creating new partitions, you can use the partitions in a ring structure, so that new data rotates around the partition scheme one partition at a time.

For example, suppose you want to keep the last month's worth of data, partitioned by day. You could create 31 fixed partitions, where the partition number represents the day of the month. You could create a persisted calculated column that computes the day of the month based on a date column in the table, and partition on that computed column. For each new day's data, you switch out that day number's partition, truncate the staging table, load it with that day's data, and then switch the loaded staging table back into the partitioned table.

The major restriction on using such a scenario is that queries against the table must filter based on the partition number (the day of the month) in order to gain partition elimination. If users issue queries and filter by the original date column, the queries do not perform well because SQL Server's optimizer cannot eliminate inapplicable partitions.

So in such rotating windows, it is usually required that users issue queries through some kind of controlled mechanism: the application code or a stored procedure. When the user's query reaches SQL Server, any of the original date references are translated into a filter on the day of the month, and partition elimination occurs.

As a particular example, Microsoft Sales has a database that contains a table that contains the last 31 days worth of data, partitioned by hour. Each partition represents the hour of the month. The total number of partitions is 31*24 = 744 partitions. The partition function is fixed and no SPLIT or MERGE operations occur. The newest data is loaded into the current hour by staging data 30 seconds at a time (using the two staging tables scenario described below in "Continuous Loading".) At the top of the hour, the oldest partition is switched out and truncated (because downstream processes have already extracted their data) and becomes the newest

47

hour's partition. If the time now is 2008-06-10 09:59:00, it will be the 225th hour of the month of June 2008, and we will be loading data into partition 225. Partition 226 is the oldest partition, containing data for the 226th hour of May (2008-05-10, 10:00:00). At the top of the hour, the data from partition 226 is switched out and the table truncated, and new data will be inserted into partition 226. (I am indebted to LeRoy Tuttle for this example.)

Data Loading Scenarios

The most efficient way of loading new data into a partitioned table is to load a staging table first and then use the SPLIT and SWITCH operations to switch the staging table into the partitioned table. (Partitioning can be key to improving data loads. For example, at the SQL Server 2008 Launch in February, SQL Server 2008 achieved the world record for ETL loading into a partitioned table. See "ETL World Record!" at http://blogs.msdn.com/sqlperf/archive/2008/02/27/etl-world-record.aspx.)

Periodic Loading into a Relational Data Warehouse

In data warehouse scenarios, loading data on a periodic basis is not usually a problem, because it is common to extract, transform, and load data over periods of time from multiple data sources and stage the data before making it available for end users. For a partitioned table, the final staging table should be in the target database, and for a switch to occur, the staging table must reside on the same filegroup as the target partition.

In such a scenario, the steps are similar to that of loading the leading edge of a partitioned table using a sliding window. Two additional factors that can help data loading are:

- Use bulk loading techniques to load the staging table.
- Consider using multiple files on filegroups.

For more information, see the TechNet white paper "Loading Bulk Data into a Partitioned Table" at http://technet.microsoft.com/en-us/library/cc966380.aspx.

Continuous Loading

In an OLTP scenario, new data may be coming in continuously. If users are querying the newest partition as well, inserting data continuously may lead to blocking: User queries may block the inserts, and similarly, inserts may block the user queries.

Contention on the loading table or partition can be reduced by using snapshot isolation—in particular, the READ COMMITTED SNAPSHOT isolation level. Under READ COMMITTED SNAPSHOT isolation, inserts into a table do not cause activity in the **tempdb** version store, so the **tempdb** overhead is minimal for inserts, but no shared locks will be taken by user queries on the same partition.

In other cases, when data is being inserted into a partitioned table continuously at a high rate, you may still be able to stage the data for short periods of time in staging tables and then insert that data into the newest partition repeatedly until the window for the current partition passes and data then is inserted into the next partition. For example, suppose you have two staging tables that receive 30 seconds worth of data each, on an alternate basis: one table for the first half of a minute, the second table for the second half of a minute. An insert stored procedure determines which half of the minute the current insert is in, and then it inserts into the first staging table. When 30 seconds is up, the insert procedure determines it must insert into the second staging table. Another stored procedure then loads the data from the first staging table into the newest partition of the table, and then it truncates the first staging table. After another

30 seconds, the same stored procedure inserts the data from the second stored procedure and puts it into the current partition, and then it truncates the second staging table.

Handling Large Numbers of Partitions

The limit on the number of table partitions is 1,000, meaning that the maximum number of boundary values in the partition function is 999. In some cases you can exceed that number. Suppose for example that you want to partition the most recent data by day: The maximum of 1,000 partitions implies that you can keep just less than three years of data in a partitioned table. That may not be acceptable, in which case you have a number of options:

- Consolidate the oldest partitions periodically. You might decide to consolidate the older data into a partition time period that is longer than the standard. For example, you might partition the current year by day, the previous year by month, and so on. In that case you must incur some penalty of data movement, because as you age the oldest finer-grained partition into the larger partition, somehow you must get the data from one partition into the other. For example, you could switch out the oldest day's worth of data, and then directly insert it into the broader last year's partition. Assuming the data in the older partition is read-only or mostly read-only, the inserts might not cause contention with user queries on the data. Alternatively, you could switch both the last daily partition and last year's partition out, each into their own staging tables, and then copy the oldest day's data into last year's partition, and switch the result back into the table. The overall effect is to maintain a sliding window for the current year's data and maintain a wider partition for the older year's data.

- Partitioned views or federated servers. If you simply must maintain more than 1,000 partitions, you could maintain more than one partitioned table, with a partitioned view declared over the partitioned tables. If the two tables are kept in separate databases on separate servers, you could create a distributed partitioned view across two SQL Server instances. For more information, see the topics related to "Using Partitioned Views" and "Federated Database Servers" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms190019.aspx and http://msdn.microsoft.com/en-us/library/ms190381.aspx, respectively.

Consolidating Partitions

You may need to consolidate partitions. However, removing a boundary value on nonempty partitions in a partitioned table involves data movement and significant additional activity in the transaction log.

If you need to consolidate two neighboring partitions, if they contain data that is primarily read only, and if their data can be missing for a short period, you can switch each of them out to staging tables and then remove the boundary value separating them. You can then insert one staging table's data into the staging table on the target partition's filegroup, and then switch that data back into the partitioned table. Users can continue to query the partitioned table, although they cannot see any of the consolidated data until the switch process is completed.

Another approach to consolidating two neighboring partitions that contain data is to extract the rows from each partition into a stand-alone unpartitioned staging table that resides on the desired target filegroup. Add the appropriate indexes and constraint to the staging table. Then switch out the two partitions, perform the MERGE removing the boundary value between the two empty partitions, and switch in the staging table.

49

Both these approaches require that update activity (insert, update, and delete) be prevented on the neighboring partitions for a certain period of time until the MERGE and SWITCH operations can be completed.

Maintaining Non-Time-Based Ranges

Though partitioning on time is by far the most common use for table partitioning, it is possible to partition on data types other than time. For example, you might partition on location, product, type, and so on. This can still deliver benefits for management and query performance, provided that queries actually filter based on the partition column. However, the data type must be chosen carefully: Strings in particular are handled by the partition function based on the collation order of characters, and the collation order is case-sensitive (see "Other Data Types as Partition Boundaries" above.) The challenge is to choose boundary values that will more or less balance out the data distribution so that the partition elimination works well.

It's important to plan for growth so that new boundary values can be added with a minimum of data movement. You can use the $PARTITION function to test a partition function and see how the data would be distributed. For an example, see "Using the $PARTITION Function" above.

Archiving Data Out of a Partitioned Table

When you switch out older data from a partitioned table, you may need to archive that data. Depending upon application needs, you may need to keep the archived data in the same database, or in a separate archive database, which can be on the same server or a different archive server.

Archiving Within the Same Database

If you are archiving to data in the same database, you can use partition switching to archive the oldest partition of a partitioned table. If you keep the archived data in a partitioned table, you can switch the oldest partition out of the partitioned table to the newest partition in the archive table. The data never leaves its filegroup, and the process is very fast.

For example, the **AdventureWorks2008** sliding window example provided for SQL Server 2008 on CodePlex switches the oldest month's data from the Production.TransactionHistory table into the newest partition of the Production.TransactionHistoryArchive table. After the switch, the **AdventureWorks2008** script merges the oldest partitions in the archive table causing data movement.

You can avoid the data movement in the archive table by partitioning it with the same granularity that the main partitioned table has, and then switching out the very last archive partition, storing the data to tape, dropping the staging table that has the oldest data, and dropping the staging table and reusing the filegroup.

Some considerations are listed here:

- The most efficient way to delete old partitioned data from the database is to switch the data out to a staging table, optionally export the data for archival storage, and then truncate the table.
- If you are archiving to a different database or different server, you can still switch out the oldest partition to an archive table and then export the data from the archive table. However, the filegroup used by the archive table belongs to the original database, and it cannot be moved to another database. After the archive table is exported, you can truncate the table and use the filegroup again for other purposes.

- If you use dedicated filegroups for archived data and the archived data is read-only, you can mark the filegroup read-only and reduce the locks taken on the tables when archived data is queried. However, you cannot mark a filegroup as read-only with other users in the database.
- You can store archived data on slower disks, provided you move the filegroup and file: For more information, see the next section.

Moving Older Partitions to Slower Disks

Often the older data in a table partitioned by date is queried less often, and queries against the older data may not require as quick a response time as the newer. In such cases you may be able to keep the older partitions on less expensive larger disks, reducing the overall costs of storage. As the partitions move forward in time in a sliding window scenario, there may be a transition point where you can move a partition periodically from a faster to slower set of disks.

Suppose you have your partitioned table spread across two groups of disks, a set of relatively faster disks and another slower and less expensive set. The older partitions are kept on filegroups that reside on the slower set. The basic task is to take the oldest partition from the faster disk set and then re-create it as a new partition on the slower set. The basic strategy is the following:

1. Copy the partition's data out of the table and onto the target filegroup on the slower disk set.
2. Remove the partition on the faster disk set using MERGE.
3. Add a new partition with the same constraint to the slower disk set using SPLIT.
4. Move the data in using SWITCH.

The details and complexity arise with how the data should be copied out. One option is to switch the data out to a table on the faster disk set and then copy that data over to the slower filegroup. However, the data in that partition will be unavailable during the copy period. Another option is to copy the data directly from the partition to a staging table on the slower disk group first, switch the data out of the partition on the faster disks, merge the partition, use SPLIT to create a new partition on the slower disk group, and then switch the copied data into the new partition. This latter approach works well when the data is used for read-only, and it results in the data being more available for querying.

The following illustration uses the second approach. Consider a table with six partitions, half of the partitions on filegroups placed on a fast disk set, and the other half placed on slow disks. Figure 24 illustrates the starting point.

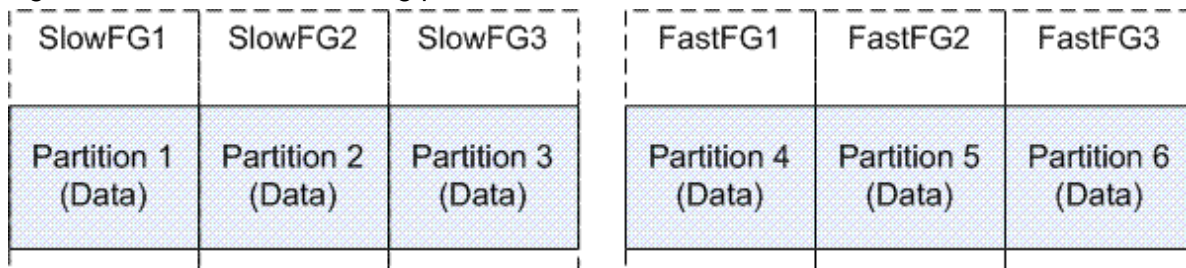| SlowFG1	SlowFG2	SlowFG3		FastFG1	FastFG2	FastFG3
Partition 1 (Data)	Partition 2 (Data)	Partition 3 (Data)		Partition 4 (Data)	Partition 5 (Data)	Partition 6 (Data)

Figure 24. A partitioned table spread across slower and faster disk sets

The older partitions 1, 2, and 3 are all on a slower, less expensive disk set. The newer partitions 4, 5, and 6 are on a faster, higher performance disk set. We want to take partition 4 and move it to the slower disk set:

51

1. First, prepare a new filegroup on the slower disk set, which will be the target location for the new partition 4.
2. Next, build a staging table on the new filegroup and copy the data to it. This could be done by exporting the partition's data to disk and bulk loading into the staging table.
3. After the copy, add the appropriate indexes and constraint on the staging table for partition 4's data.
4. Create a temporary staging table on the current partition 4's filegroup to serve as a place to switch data out of partition 4.
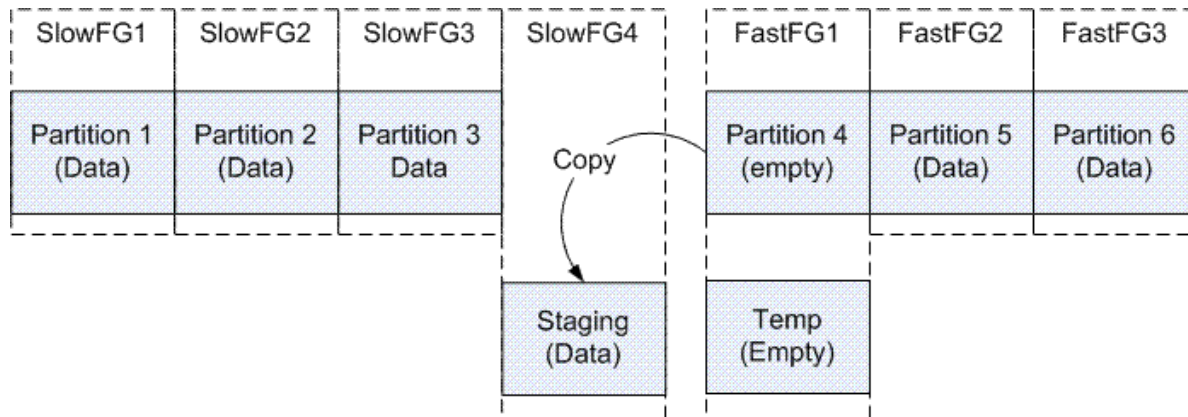
These steps are illustrated in Figure 25.

Figure 25. Manually copying the staging table to a slower disk filegroup

After the data has been copied, the next step is to switch out partition 4's data to the temporary location and merge partition 4. The result is shown in Figure 26.
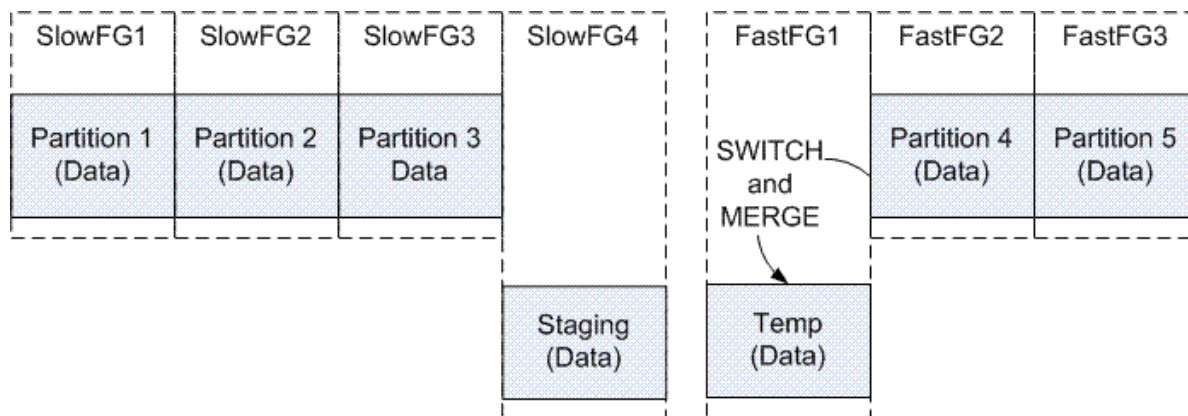

Figure 26. The result of SWITCH and MERGE operations on the fast filegroup

Note that the partitions have been renumbered. The SWITCH and MERGE operations are metadata-only operations, but for this short period of time, the data for the partition range specified by the old partition 4 is not available for querying.

The next step is to split the partition function creating a new partition 4 on the slower disk set and then switch the data from the staging table into it:

1. Assign the NEXT USED property to the SlowFG4 filegroup.

2. Add the boundary value for the old partition 4 back into the partition function using SPLIT.
3. Switch the data from the staging table into the new partition.

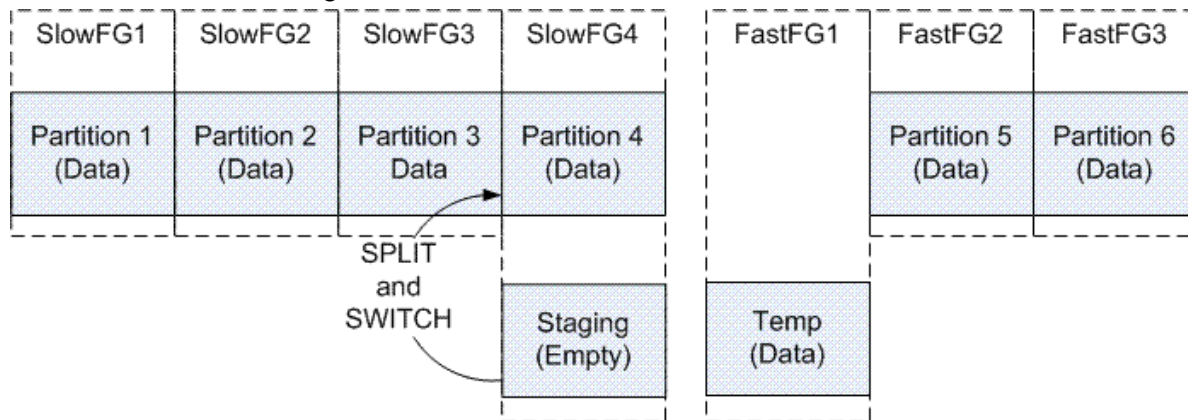The result is shown in Figure 27.

Figure 27. Switching the staging table back into partition 4 on a slower-disk filegroup

Both the split to create the new partition 4 and the switch from the staging table into the partition are metadata-only operations. Because all necessary indexes and the table constraint were already completed on the staging table, the switch needs no other preparation and completes the process. At this point, deleting the temporary staging table leaves filegroup FastFG1 empty and available for other use.

Table Partitioning Best Practices

Table partitioning is useful on very large data tables for primarily two reasons.

The major reason for partitioning is to gain better management and control of large tables by partitioning them. To gain better management of large tables, you can:

- Rebuild and reorganize indexes by partition.
- Use partition-aligned indexed views in switching operations.
- Use a sliding window strategy for quickly bringing in new data and archiving old data.

Additionally, SQL Server's query optimizer can use partition elimination and parallelism to increase appropriately filtered query performance against partitioned tables. To make use of partition elimination:

- Ensure that indexes are aligned with the partitioned table, and that indexed views are partition-aligned.
- Ensure that queries against the partitioned tables have filters based on the partition column.
- On data warehouse joins, keep the join column simple (such as an integer or date) and explicit, so as to take advantage of bitmap filtering for star joins.

In general, to take full advantage of table partitioning, you should:

53

- Make sure that the configuration of **max degree of parallelism** is set sufficiently high to take advantage of parallel operations, or else add a MAXDOP query hint to fine-tune the degree of parallelism.
- Maintain an empty partition on both ends of the partitioned table and ensure that only empty partitions are split and merged in a sliding window scenario.
- Remember that RANGE RIGHT may be more convenient than RANGE LEFT in partition functions, especially when you are specifying date ranges.
- Use data types without fractional components as partition columns, such as a date or an integer.
- Always use a standard language-independent date format when specifying partition function boundary values.
- Use an integer-based date and date dimension in data warehouses.
- Use a single column of the table as the partitioned column whenever possible. If you must partition across more than one column, you can use a persisted computed column as the partitioning column. But then to achieve partition elimination, you must control queries to ensure they reference the partition column in their filters.
- Use SWITCH with MERGE to drop partition data: Switch out the partition and remove the partition's boundary value using MERGE.
- Use TRUNCATE TABLE to delete partition data by switching a partition out to a staging table and truncating the staging table.
- Check for partition elimination in query plans.
- Place read only data on read-only filegroups to reduce locking and simplify recovery for piecemeal restores.
- Spread filegroups across all disks for maximum I/O performance.
- Automate sliding window scenarios using available tools.

Essential reading for table partitioning best practices is the white paper "Top 10 Best Practices for Building A Large Scale Data Warehouse" on the SQL Customer Advisory Team Web site at:

http://sqlcat.com/top10lists/archive/2008/02/06/top-10-best-practices-for-building-a-large-scale-relational-data-warehouse.aspx

Even though the focus of the paper is on relational data warehouses, most of the recommendations are relevant for table partitioning in an OLTP context as well.

Conclusion

Table partitioning addresses the challenges of querying and managing large tables. When you partition a table or index, many operations can apply at the partition level rather than the table level. Partition elimination allows the query processor to eliminate inapplicable partitions from a query plan, and tables and indexes can often be managed at the partition level. Partitioning a table requires careful analysis to choose the appropriate partition column, boundary values for a partition function, and filegroup placement for the partition scheme. The result is a method for managing large data tables in both OLTP and relational data warehouse scenarios.

In SQL Server 2008, indexed views can be partition-aligned, so that they participate in the SWITCH operation. You can also use new features of SQL Server 2008, such as data compression and finer-grained lock escalation control, at the partition level. In addition,

Microsoft provides tools to assist in automating changes to partitioned tables over time, in particular for the sliding window scenario.

Appendix: Inspecting Table Partitioning Objects

The following sections contain queries of table partitioning metadata that you can use to inspect partition functions, partition schemes, and partitioned tables.

Inspecting a Partition Function

You can inspect any partition function by using four catalog views. Figure A1 shows an Entity-Relationship diagram of the four catalog views, considered as entities.
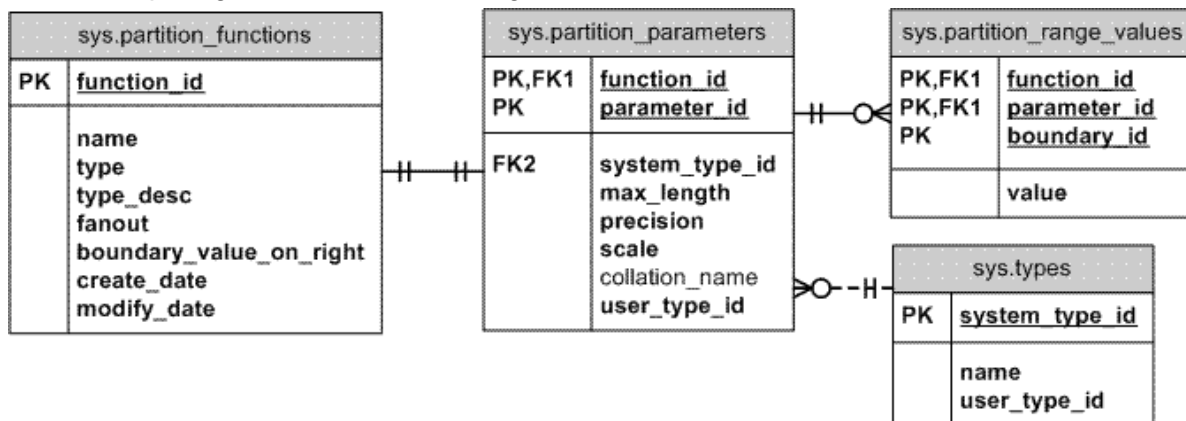

Figure A1. Catalog views for partition functions

Full descriptions of these catalog views can be found in SQL Server 2008 Books Online. For purposes of querying these views, among the more useful columns are:

- *Sys.partition_functions* returns one row per partition function, containing function_id for joining with other catalog views, the partition function name, range direction (in boundary_value_on_right), and number of resulting partitions (in fanout).
- *Sys.partition_parameters* returns one row per partition function and parameter ID combination. (At the current time, a partition function can only have one input parameter.) Each row contains the data type ID (in system_type_id) of the input parameter, which you can join with the sys.types catalog view to get the data type name. It also contains:
    - The length, precision, and scale of the input parameter (if required by the data type).
    - The *collation_name* of a character-based data type.
    - A user_type_id, implying future support for more than native SQL Server data types as partition function input parameters.
- *Sys.partition_range_values* returns one row per boundary value of the partition function, assigning a boundary_id and value.

To extract general information about all partition functions, without the explicit boundary values, you can use the following steps:

1. Select the desired columns from sys.partition_functions.

2. Join with sys.partition_parameters to get the value for system_type_id.
3. Join with sys.types to get the data type name of the input parameter.

The following code shows an example of these steps.

```
SELECT
    PF.function_id
  , PF.name
  , PF.fanout AS NumPartitions
  , CASE WHEN PF.boundary_value_on_right = 0
      THEN 'LEFT' ELSE 'RIGHT' END AS RangeType
  , PP.parameter_id
  , CASE WHEN PP.system_type_id = PP.user_type_id
            THEN T1.name ELSE T2.name END AS ParameterDataType
FROM sys.partition_functions AS PF
JOIN sys.partition_parameters AS PP
      ON PF.function_id = PP.function_id
JOIN sys.types AS T1
      ON T1.system_type_id = PP.system_type_id
JOIN sys.types AS T2
      ON T2.user_type_id= PP.user_type_id;
```

To query for all information about a partition function, including partitions and boundaries, you can use the above steps and add the following:

4. Join with partition_range_values to get each boundary values.

The following query shows the boundary values in addition to basic function information.

```
SELECT
    PF.function_id
  , PF.name
  , PF.fanout AS NumPartitions
  , CASE WHEN PF.boundary_value_on_right = 0
      THEN 'LEFT' ELSE 'RIGHT' END AS RangeType
  , PP.parameter_id
  , CASE WHEN PP.system_type_id = PP.user_type_id
            THEN T1.name ELSE T2.name END AS ParameterDataType
  , PRV.boundary_id
  , PRV.value
  , CASE WHEN PF.boundary_value_on_right = 0
      THEN PRV.boundary_id ELSE PRV.boundary_id + 1 END AS PartitionNumber
```

```
FROM sys.partition_functions AS PF
JOIN sys.partition_parameters AS PP
     ON PF.function_id = PP.function_id
JOIN sys.types AS T1
     ON T1.system_type_id = PP.system_type_id
JOIN sys.types AS T2
     ON T2.user_type_id= PP.user_type_id
JOIN sys.partition_range_values AS PRV
     ON PP.function_id = PRV.function_id
   AND PP.parameter_id = PRV.parameter_id;
```

Partition functions are database-wide objects, meaning that they are stored within a database. You can script a partition function by using Object Explorer in SQL Server Management Studio, going to the Storage node of a database, expanding the Partition Functions node and then right-clicking the partition function. However, partition functions are not listed as database objects in the sys.all_objects or sys.objects system tables.

You can also script a partition function using the Windows PowerShell™ command-line interface from SQL Server Management Studio, as you can see in the following script. For instance, to script out the PF2_Right() partition function, start by right-clicking the database name in Object Explorer to invoke SQL Server PowerShell. In the resulting window, execute the following, one line at a time.

```
cd PartitionFunctions
$pso = New-Object Microsoft.SqlServer.Management.Smo.ScriptingOptions
(get-item PF2_Right).Script($pso)
```

Inspecting a Partition Scheme

You can view partition schemes using catalog views. The relevant catalog views are sys.partition_schemes, sys.destination_data_spaces, and sys.filegroups. Figure A2 shows these catalog views interpreted as entities.
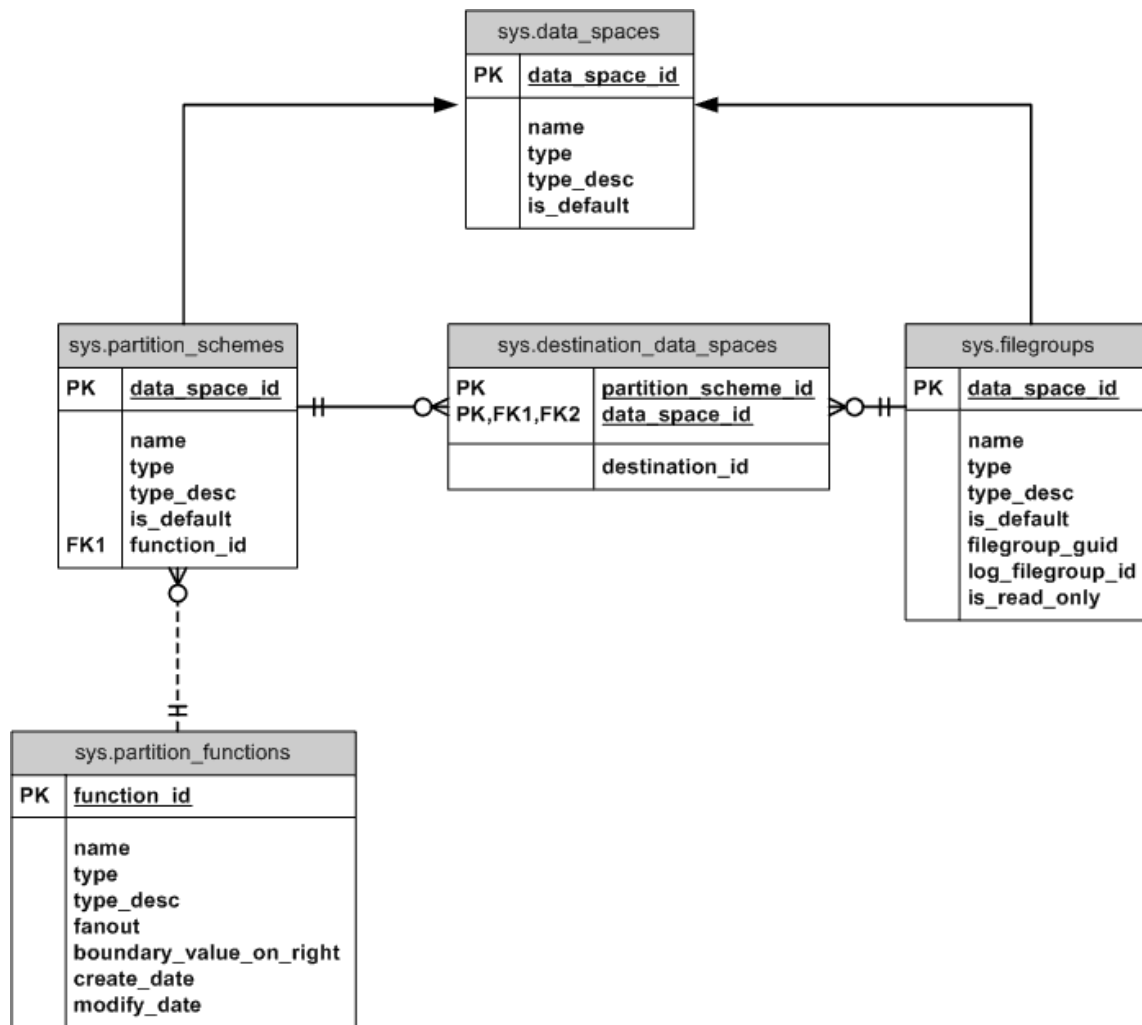
Figure A2. Catalog views for partition schemes

Both sys.partition_schemes and sys.filegroups inherit columns from sys.data_spaces. sys.data_spaces includes information for both partition schemes and filegroups. sys.partition_functions adds function_id to identify the function, and it adds some additional filegroup information for sys.filegroups. sys.partition_schemes does not contain a partition scheme ID as such, but the data_space_id maps to the partition_scheme_id of sys.destination_data_spaces.

To query the catalog views:

- Get the partition scheme name from sys.partition_schemes.name.
- Join sys.partition_schemes with sys.destination_data_spaces on data_space_id and partition_scheme_id, respectively.
- Get the filegroup name from sys.filegroups.
- Join sys.filegroups with sys.destination_data_spaces on data_space_id.

The following code queries the above tables and returns information for the partition scheme PS_Right, with Filegroup5 designated as the NEXT USED filegroup.

```
SELECT

        SPS.name AS PartitionSchemeName

      , SDD.destination_id

      , SF.name AS FileGroup

FROM sys.partition_schemes AS SPS

JOIN sys.destination_data_spaces AS SDD

      ON SPS.data_space_id = SDD.partition_scheme_id

JOIN sys.filegroups AS SF

      ON SF.data_space_id = SDD.data_space_id

WHERE SPS.name = 'PS2_Right';
```

If there is a NEXT USED filegroup, the marked filegroup gets the next ordinal number in destination_id.

The sys.destination_data_spaces destination_id stores the partition number, except for the NEXT USED filegroup. The filegroup marked as NEXT USED has a number that is one greater than the number of partitions defined by the partition function. Because the number of partitions is defined in the fanout column of sys.partition_functions, you can use it to determine what the NEXT USED filegroup is, as well as what the actual partition IDs are.

```
SELECT

        SPS.name AS PartitionSchemeName

      , SPF.name AS PartitionFunctionName

      , CASE WHEN SDD.destination_id <= SPF.fanout THEN SDD.destination_id

            ELSE NULL END AS PartitionID

      , SF.name AS FileGroup

      , CASE WHEN SDD.destination_id > SPF.fanout THEN 1

            ELSE 0 END AS NextUsed

FROM sys.partition_schemes AS SPS

JOIN sys.partition_functions AS SPF

      ON SPF.function_id = SPS.function_id

JOIN sys.destination_data_spaces AS SDD

      ON SPS.data_space_id = SDD.partition_scheme_id

JOIN sys.filegroups AS SF

      ON SF.data_space_id = SDD.data_space_id

WHERE SPS.name = 'PS2_Right';
```

You can complete the query by showing the actual boundary values defined by the partition function, so that you can match up the function's values with the filegroups that the data would be stored in.

```
SELECT

        SPS.name AS PartitionSchemeName
```

```sql
        , CASE WHEN SDD.destination_id <= SPF.fanout THEN SDD.destination_id
              ELSE NULL END AS PartitionID
        , SPF.name AS PartitionFunctionName
        , SPRV.value AS BoundaryValue
        , CASE WHEN SDD.destination_id > SPF.fanout THEN 1
              ELSE 0 END AS NextUsed
        , SF.name AS FileGroup
FROM sys.partition_schemes AS SPS
JOIN sys.partition_functions AS SPF
        ON SPS.function_id = SPF.function_id
JOIN sys.destination_data_spaces AS SDD
        ON SDD.partition_scheme_id = SPS.data_space_id
JOIN sys.filegroups AS SF
        ON SF.data_space_id = SDD.data_space_id
LEFT JOIN sys.partition_range_values AS SPRV
        ON SPRV.function_id = SPF.function_id
        AND SDD.destination_id =
CASE WHEN SPF.boundary_value_on_right = 0 THEN SPRV.boundary_id
        ELSE SPRV.boundary_id + 1 END
WHERE SPS.name = 'PS2_Right';
```

In the above query:

- The sys.partition_functions fanout column is used to get the number of partitions.
- If the sys.destination_data_spaces destination_id is less than or equal to the number of partitions, it is the partition ID.
- If the sys.destination_data_spaces destination_id is greater than the number of partitions, that is the NEXT USED filegroup.
- If the partition function is RANGE LEFT, the value of boundary_id from sys.partition_range_values is the same as the partition ID; for RANGE RIGHT, just add 1 to the boundary_id to get the partition ID.

You can script a partition scheme by using Object Explorer in SQL Server Management Studio, clicking the Storage node of a database, expanding the Partition Schemes node, and then right-clicking the partition scheme. You can also use SQL Server PowerShell to retrieve a partition scheme script by right-clicking the database name and issuing the following PS SQL commands.

```
cd PartitionSchemes
$pso = New-Object Microsoft.SqlServer.Management.Smo.ScriptingOptions
(get-item PS2_Right).Script($pso)
```

60

Inspecting a Partitioned Table

Every table in SQL Server 2005 and SQL Server 2008 has at least one row in the sys.partitions catalog view, so in an extended sense they could all be considered partitioned. But more exactly, we will only consider tables that are linked with a partition scheme as partitioned.

The key to determining whether a table is partitioned is the table (or index) data_space_id in the sys.indexes catalog view, and whether it has an associated partition scheme in the sys.data_spaces catalog view. All tables that are placed on a partition scheme will have 'PS' (for partition scheme) as the type for their data_space_id in sys.data_spaces.

Even if you have a partitioned table and remove all the boundary values of its partition function using the MERGE option, the result will be a table with just one partition. However, the table will still be defined using a partition scheme, and you can add boundaries and partitions back into it using SPLIT.

The quickest way to query whether a table has a link to a partition scheme is to inspect the sys.indexes catalog view for the row corresponding to the table (whether a heap or clustered), find its data_space_id, and then find a matching partition scheme in the sys.data_spaces table. Figure A3 shows the relationship of the metadata tables you can use to find partitioned table information.
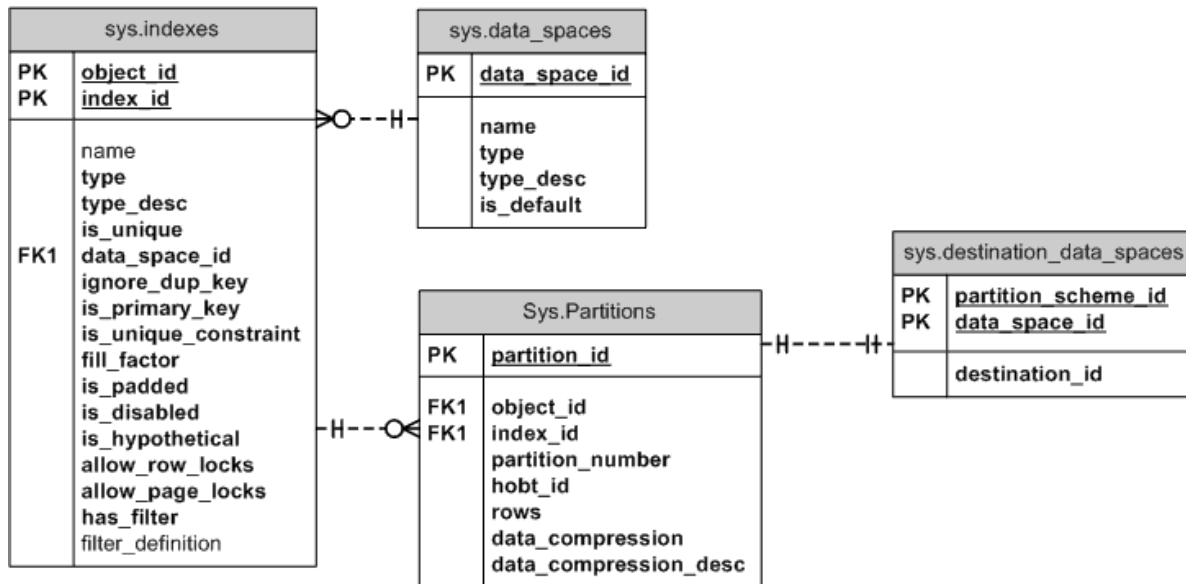

Figure A3. Catalog views for partitioned tables

To just get a list of all partitioned tables (that is, tables that are created on a partition scheme), and their partition function, you can join the sys.partitions, sys.indexes, and sys.data_spaces catalog views.

```
SELECT
        OBJECT_NAME(SI.object_id) AS PartitionedTable
        , DS.name AS PartitionScheme
FROM sys.indexes AS SI
JOIN sys.data_spaces AS DS
ON DS.data_space_id = SI.data_space_id
```

```
WHERE DS.type = 'PS'

AND OBJECTPROPERTYEX(SI.object_id, 'BaseType') = 'U'

AND SI.index_id IN(0,1);
```

The table name is available from the object_id column of sys.indexes, and the name of the partition scheme from the name column of sys.data_spaces. If a row in sys.data_spaces has a type of 'PS', it is a partition scheme. The query returns data for just base tables (and not system tables) by using the OBJECTPROPERTYEX() system function. Finally, the sys.indexes catalog view also contains multiple rows for indexes related to the table, so limiting the query to an index_id of 0 or 1 returns information for just heap or clustered tables.

To add the partition function, just add joins to sys.partition_schemes and sys.partition_functions.

```
SELECT

      OBJECT_NAME(SI.object_id) AS PartitionedTable

      , DS.name AS PartitionScheme

      , PF.name AS PartitionFunction

FROM sys.indexes AS SI

JOIN sys.data_spaces AS DS

      ON DS.data_space_id = SI.data_space_id

JOIN sys.partition_schemes AS PS

      ON PS.data_space_id = SI.data_space_id

JOIN sys.partition_functions AS PF

      ON PF.function_id = PS.function_id

WHERE DS.type = 'PS'

AND OBJECTPROPERTYEX(SI.object_id, 'BaseType') = 'U'

AND SI.index_id IN(0,1);
```

Given a partitioned table, it is useful to know the filegroups that its partitions reside on. The following query starts with the sys.partitions catalog view to show individual table partitions with the table name, partition scheme, partition function, partition number, and approximate rows per partition.

```
SELECT

      OBJECT_NAME(SI.object_id) AS PartitionedTable

      , DS.name AS PartitionScheme

      , PF.name AS PartitionFunction

      , P.partition_number

      , P.rows

FROM sys.partitions AS P

JOIN sys.indexes AS SI

      ON P.object_id = SI.object_id AND P.index_id = SI.index_id

JOIN sys.data_spaces AS DS
```

```
        ON DS.data_space_id = SI.data_space_id
JOIN sys.partition_schemes AS PS
        ON PS.data_space_id = SI.data_space_id
JOIN sys.partition_functions AS PF
        ON PF.function_id = PS.function_id
WHERE DS.type = 'PS'
AND OBJECTPROPERTYEX(SI.object_id, 'BaseType') = 'U'
AND SI.type IN(0,1);
```

The number of rows from sys.partitions is considered approximate. Additional columns are available in the sys.partitions catalog view for data compression information. (See "Querying Data and Metadata from Partitioned Tables and Indexes" in SQL Server 2008 Books Online at http://msdn.microsoft.com/en-us/library/ms187924.aspx.)

Next, expand the query to get the filegroups for each partition number in a partitioned table.

```
SELECT
        OBJECT_NAME(SI.object_id) AS PartitionedTable
      , DS.name AS PartitionSchemeName
      , PF.name AS PartitionFunction
      , P.partition_number AS PartitionNumber
      , P.rows AS PartitionRows
      , FG.name AS FileGroupName
FROM sys.partitions AS P
JOIN sys.indexes AS SI
        ON P.object_id = SI.object_id AND P.index_id = SI.index_id
JOIN sys.data_spaces AS DS
        ON DS.data_space_id = SI.data_space_id
JOIN sys.partition_schemes AS PS
        ON PS.data_space_id = SI.data_space_id
JOIN sys.partition_functions AS PF
        ON PF.function_id = PS.function_id
JOIN sys.destination_data_spaces AS DDS
        ON DDS.partition_scheme_id = SI.data_space_id
        AND DDS.destination_id = P.partition_number
JOIN sys.filegroups AS FG
        ON DDS.data_space_id = FG.data_space_id
WHERE DS.type = 'PS'
AND OBJECTPROPERTYEX(SI.object_id, 'BaseType') = 'U'
AND SI.type IN(0,1);
```

63

Add a join with the sys.destination_data_spaces catalog view on two columns: Join the partition number of sys.partitions and the destination_id of sys.destination_databases; also join the sys.indexes data_space_id (which is the partition scheme ID) with the partition_scheme_id of sys.destination_data_spaces. To match up filegroups, join data_space_id between sys.filegroups and sys.destination_data_spaces.

Inspecting Partitioned Indexes

Index alignment occurs automatically when you place the partition column in the secondary nonunique index key or add it as an included column. The end result is that the indexes become partitioned and are assigned rows in the sys.partitions table. You can tell whether an index is partitioned using the same technique for detecting whether a table is partitioned: Find the data_space_id in the sys.indexes catalog view and join with the sys.data_spaces catalog view to determine whether it is placed on a partition scheme.

```
SELECT

      OBJECT_NAME(SI.object_id) AS PartitionedTable

      , SI.name

      , SI.index_id

      , DS.name AS PartitionScheme

FROM sys.indexes AS SI

JOIN sys.data_spaces AS DS

ON DS.data_space_id = SI.data_space_id

WHERE DS.type = 'PS';
```

This query shows only the partitioned indexes that are placed on a partition scheme. To see both aligned and nonaligned indexes for a given table, just add the type column from sys.data_spaces, remove the requirement for the 'PS' type, and filter by the table name.

```
SELECT

      OBJECT_NAME(SI.object_id) AS PartitionedTable

      , SI.name

      , SI.index_id

      , DS.type

      , DS.name AS PartitionScheme

FROM sys.indexes AS SI

JOIN sys.data_spaces AS DS

ON DS.data_space_id = SI.data_space_id

WHERE OBJECT_NAME(SI.object_id) = 'PartitionedTable'
```

To see all the partitions for partitioned tables and indexes, you can remove the changes to the first query and add a join with sys.partitions.

```
SELECT

      OBJECT_NAME(SI.object_id) AS PartitionedTable

      , DS.name AS PartitionScheme
```

```sql
      , SI.name
      , SI.index_id
      , SP.partition_number
      , SP.rows
FROM sys.indexes AS SI
JOIN sys.data_spaces AS DS
      ON DS.data_space_id = SI.data_space_id
JOIN sys.partitions AS SP
      ON SP.object_id = SI.object_id
AND SP.index_id = SI.index_id
WHERE DS.type = 'PS'
ORDER BY 1, 2, 3, 4, 5;
```

The ORDER BY clause helps when you are viewing the partition numbers and row counts, to make sure they match up.

In SQL Server 2008, indexed views can also participate in table partitioning (see the section "Partition-Aligned Indexed Views" above).