At a glance:
Creating partitioned tables
Adding and merging partitions
Creating and managing partitioned indexes

Simplify database maintenance with table partitions

Noah Gomez

In the past, database administrators managing tables containing millions of rows of data were forced to create multiple tables. Once those tables were partitioned, the administrator had to tie the tables back together during the execution of many of their queries. Tying partitions together involved creating a partitioned view or a wrapper stored procedure that figured out where the data lived and executed another stored procedure to hit only the partitions needed to return the dataset.

While these methods worked, they were burdensome. The administration of multiple tables and their indexes, as well as the methods used for tying the tables back together often caused administration and maintenance issues. In addition, creating multiple tables to partition data led to a degree of inflexibility, as the stored procedures, maintenance jobs, Data Transformation Services (DTS) jobs, application code, and other processes had to understand the nature of the partitioning. So to allow you to add or drop these quasi partitions without changing your code, these elements were typically created in a non-dynamic manner, and as a result they were inefficient.

The Enterprise and Developer editions of SQL Server 2005 let you partition large amounts of data contained in a single table into multiple smaller partitions that can be managed and maintained more effectively. The ability to create data segments that are accessed through a single point-of-entry reduces many of the administration issues that came with the old way of doing things. Using a single point of entry (table name or index name) hides the multiple data segments from the application code and allows the administrator or developer to change the partitions as necessary without having to adjust the code base.

Figure 1 Assigning filegroups to a partition scheme

```
--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

--Place all partitions into the different filegroups
CREATE PARTITION SCHEME Different_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      TO (Filegroup1, Filegroup2, Filegroup3, Filegroup4)
--Filegroups must currently exist in database

--Place multiple partitions into the different filegroups
CREATE PARTITION SCHEME Multiple_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      TO (Filegroup1, Filegroup2, Filegroup1, Filegroup2)
--Filegroups must currently exist in database
```

In short, you can create multiple partitions, move those partitions around, drop old partitions, and even change the way data is partitioned without ever having to adjust the code in your application. Your application code simply continues to call the same base table or index name. Meanwhile, you can reduce the amount of data that individual indexes contain, in turn decreasing maintenance times for those indexes, and you can increase data load speed by loading into empty partitions.

Technically, every SQL Server 2005 table is partitioned – every table has at least one partition. What SQL Server 2005 does is allow database administrators to create additional partitions on each table. Table and index partitions are hard-defined, row-level partitions (partitioning by columns is not allowed) that allow a single point of entry (table name or index name) without the application code needing to know the number of partitions behind the point of entry. Partitions can exist on the base table as well as the indexes associated with the table.

Creating partitioned tables

You use partition functions and partition schemes to create a table that has the ability to grow beyond the default single partition. These objects are what allow you to divide the data into specific segments and control where those data segments are located in your storage design. You can, for example, spread data out over multiple drive arrays based on the age of the data or using other common differentiators. Note that a table can be partitioned based on one column of the table and each partition must contain data that cannot be placed into the other partitions.

Partition functions When partitioning a table, the first decision is how you want to divide the data into different segments. A partition function is used to map the individual rows of data into the different partitions. Those individual rows of data can be mapped by any type of column except for text, ntext, image, xml, timestamp, varchar(max), nvarchar(max), varbinary(max), alias data types, or common

Figure 2 Placing rows of data and viewing the distribution

```
--Prepare database
IF OBJECT_ID('Partitioned_Table') IS NOT NULL
DROP TABLE Partitioned_Table
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Primary_Left_Scheme')
DROP PARTITION SCHEME Primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Left_Partition')
DROP PARTITION FUNCTION Left_Partition
GO

--Create partitioned table
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
      ALL TO ([PRIMARY])

CREATE TABLE Partitioned_Table
(
col1 INT
,col2 VARCHAR(15)
) ON Primary_Left_Scheme (col1)

--Determine where values will be placed (this is not required)
--You should try to do this before executing the code
SELECT $PARTITION.Left_Partition (1)
SELECT $PARTITION.Left_Partition (2)
SELECT $PARTITION.Left_Partition (3)
SELECT $PARTITION.Left_Partition (4)
SELECT $PARTITION.Left_Partition (10)
SELECT $PARTITION.Left_Partition (11)
SELECT $PARTITION.Left_Partition (12)
SELECT $PARTITION.Left_Partition (13)
SELECT $PARTITION.Left_Partition (14)
SELECT $PARTITION.Left_Partition (100)
SELECT $PARTITION.Left_Partition (101)
SELECT $PARTITION.Left_Partition (102)
SELECT $PARTITION.Left_Partition (103)
SELECT $PARTITION.Left_Partition (104)

--Insert data into partitioned table
INSERT INTO Partitioned_Table VALUES (1,'Description')
INSERT INTO Partitioned_Table VALUES (2,'Description')
INSERT INTO Partitioned_Table VALUES (3,'Description')
INSERT INTO Partitioned_Table VALUES (4,'Description')
INSERT INTO Partitioned_Table VALUES (10,'Description')
INSERT INTO Partitioned_Table VALUES (11,'Description')
INSERT INTO Partitioned_Table VALUES (12,'Description')
INSERT INTO Partitioned_Table VALUES (13,'Description')
INSERT INTO Partitioned_Table VALUES (14,'Description')
INSERT INTO Partitioned_Table VALUES (100,'Description')
INSERT INTO Partitioned_Table VALUES (101,'Description')
INSERT INTO Partitioned_Table VALUES (102,'Description')
INSERT INTO Partitioned_Table VALUES (103,'Description')
INSERT INTO Partitioned_Table VALUES (104,'Description')

--View the distribution of data in the partitions
SELECT ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('Partitioned_Table')
```

Figure 3 Adding a partition to the left side of the function

```
--Determine where values live after new partition
SELECT $PARTITION.Left_Partition (5)   --should return a value of 2
SELECT $PARTITION.Left_Partition (1)   --should return a value of 1
SELECT $PARTITION.Left_Partition (10)  --should return a value of 2

--Add new filegroups to the partitioning scheme
ALTER PARTITION SCHEME Primary_Left_Scheme
NEXT USED [PRIMARY]

--Create new partition
ALTER PARTITION FUNCTION Left_Partition ()
SPLIT RANGE(5)

--Determine where values live after new partition
SELECT $PARTITION.Left_Partition (5)   --should return a value of 2
SELECT $PARTITION.Left_Partition (1)   --should return a value of 1
SELECT $PARTITION.Left_Partition (10)  --should return a value of 3
```

language runtime (CLR) user-defined data types. However, the partitioning function must be able to place a row of data into only one table partition – it cannot allow a row of data to belong in multiple partitions at the same time.

In order to partition a table, you must create the partitioning column in the targeted table. This partitioning column can exist in the table schema when the table is first created or you can alter the table and add the column at a later date. The column can accept NULL values but all rows containing NULL values will by default be placed in the left-most partition of the table. You can avoid this by specifying that NULL values be placed in the right-most partition of the table when you create the partitioning function. The use of the left or right partition will be an important design decision as you modify your partitioning schema and add more partitions or delete existing partitions.

When creating a partitioning function, you can choose a LEFT or RIGHT partition function. The difference between LEFT and RIGHT partitions is where the boundary values will be placed in the partitioning scheme. LEFT partitions (which is the default) include the boundary value in the partition while RIGHT partitions place the boundary value in the next partition.

To understand this concept a little better, let's look at simple LEFT and RIGHT partitions:

```
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

CREATE PARTITION FUNCTION Right_Partition (int) AS RANGE RIGHT
FOR VALUES (1,10,100)
```

In the first function (Left_Partition), the values 1, 10 and 100 are placed in the first, second and third partitions, respectively. In the second function (Right_Partition), the values are placed in the second, third and fourth partitions.

Figure 4 Moving data from one partition into another

```
--Prepare database
IF OBJECT_ID('multiple_partition') IS NOT NULL
DROP TABLE multiple_partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Primary_Left_Scheme')
DROP PARTITION SCHEME Primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Left_Partition')
DROP PARTITION FUNCTION Left_Partition
GO


--Create partitioned table
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

CREATE TABLE multiple_partition
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
) ON Primary_Left_Scheme (col1)

INSERT INTO multiple_partition VALUES (1,'Description')
INSERT INTO multiple_partition VALUES (2,'Description')
INSERT INTO multiple_partition VALUES (3,'Description')
INSERT INTO multiple_partition VALUES (4,'Description')
INSERT INTO multiple_partition VALUES (10,'Description')
INSERT INTO multiple_partition VALUES (11,'Description')
INSERT INTO multiple_partition VALUES (12,'Description')
INSERT INTO multiple_partition VALUES (13,'Description')
INSERT INTO multiple_partition VALUES (14,'Description')
INSERT INTO multiple_partition VALUES (100,'Description')
INSERT INTO multiple_partition VALUES (101,'Description')
INSERT INTO multiple_partition VALUES (102,'Description')
INSERT INTO multiple_partition VALUES (103,'Description')
INSERT INTO multiple_partition VALUES (104,'Description')

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('multiple_partition')

--Check where data would be placed
SELECT $PARTITION.Left_Partition (1)
SELECT $PARTITION.Left_Partition (10)
SELECT $PARTITION.Left_Partition (100)
SELECT $PARTITION.Left_Partition (101)

--Merge two partitions
ALTER PARTITION FUNCTION Left_Partition()
MERGE RANGE (10)

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('multiple_partition')

--Check where data would be placed
SELECT $PARTITION.Left_Partition (1)
SELECT $PARTITION.Left_Partition (10)
SELECT $PARTITION.Left_Partition (100)
SELECT $PARTITION.Left_Partition (101)
```

Figure 5 **Moving an entire table into an existing table**

```
--Prepare database
IF OBJECT_ID('multiple_partition') IS NOT NULL
DROP TABLE multiple_partition
GO

IF OBJECT_ID('single_partition') IS NOT NULL
DROP TABLE single_partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Primary_Left_Scheme')
DROP PARTITION SCHEME Primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Left_Partition')
DROP PARTITION FUNCTION Left_Partition
GO

--Create single partition table
CREATE TABLE single_partition
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
)


--Table must have a CHECK Constraint
ALTER TABLE single_partition
WITH CHECK
ADD CONSTRAINT CK_single_partition
    CHECK (col1 > 100)

INSERT INTO single_partition VALUES (101,'Description')
INSERT INTO single_partition VALUES (102,'Description')
INSERT INTO single_partition VALUES (103,'Description')
INSERT INTO single_partition VALUES (104,'Description')

--Create partitioned table
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

CREATE TABLE multiple_partition
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
) ON Primary_Left_Scheme (col1)

INSERT INTO multiple_partition VALUES (1,'Description')
INSERT INTO multiple_partition VALUES (2,'Description')
INSERT INTO multiple_partition VALUES (3,'Description')
INSERT INTO multiple_partition VALUES (4,'Description')
INSERT INTO multiple_partition VALUES (10,'Description')
INSERT INTO multiple_partition VALUES (11,'Description')
INSERT INTO multiple_partition VALUES (12,'Description')
INSERT INTO multiple_partition VALUES (13,'Description')
INSERT INTO multiple_partition VALUES (14,'Description')
INSERT INTO multiple_partition VALUES (100,'Description')

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('multiple_partition'), OBJECT_
      ID('single_partition'))

--Move the single table into the partitioned table
ALTER TABLE single_partition SWITCH TO multiple_partition PARTITION 4

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('multiple_partition'), OBJECT_
ID('single_partition'))
```

When creating a partitioned table, it's important to get the partitions as even as possible. This will help you to understand the space needed for a partition. The use of LEFT and RIGHT will determine where data is placed and this, in turn, will determine the size of the partition and the size of any indexes created on that partition.

You can determine the partition number in which a data value will be placed by using the $PARTITION function, as shown here:

```
SELECT $PARTITION.Left_Partition (10)
SELECT $PARTITION.Right_Partition (10)
```

In the first SELECT statement, the result will be 2. The second SELECT statement will return 3.

Partition schemes After you create the function and decide how your data will be divided, you must decide where the individual partitions will be created on your disk subsystem. You use partition schemes to create this disk layout. Partition schemes manage the disk storage of individual partitions by utilising filegroups to place each partition onto a location on the disk subsystem. You can configure partition schemes to have all partitions placed in a single filegroup, all partitions placed into different filegroups, or multiple partitions share filegroups. This latter method allows the database administrator a large amount of flexibility in spreading out the disk I/O.

Figure 1 shows some of the ways in which you can assign a filegroup or multiple filegroups to a partition scheme. You should be aware that the filegroups used by your partition scheme must already exist in the database before you create your partitioning scheme.

If you create the sample partition functions as shown in **Figure 1** and utilise the partition scheme to create a table, you can then determine where individual data rows are placed in your newly partitioned tables. Then you can view the distribution of those rows of data after they are inserted into your partitioned table. The code for doing all this will look something like that shown in **Figure 2**.

Modifying partitioned tables

Despite careful upfront planning, sometimes you'll need to adjust your partitioned tables after they've been created and populated. Your partition scheme may work as

intended, but you may, for example, need to add new partitions as new data is accumulated, or perhaps you may need to drop large amounts of partitioned data at one time. Fortunately, partitioned tables and the underlying partitioning structures allow for changes to be made after the table has gone live and been filled with data.

Add partitions Many partitioning plans include the ability to add a new partition at a future time. This point in time can be a particular date or it can be dependent on a value in an incremental identity column. However, if you haven't planned for this up front, you can still come in at a later date and add new partitions to a partitioned table. Consider the table created in **Figure 2**. You can add a new partition to this table to contain values greater than 500, like this:

```
--Determine where values live before new partition
SELECT $PARTITION.Left_Partition (501)
--should return a value of 4

--Create new partition
ALTER PARTITION FUNCTION Left_Partition ()
SPLIT RANGE(500)

--Determine where values live after new partition
SELECT $PARTITION.Left_Partition (501)
--should return a value of 5
```

The ability to add partitions offers great flexibility. **Figure 3** shows how you can add a partition to the left side of the function. In this case, you need to tell the partitioning scheme where to put the new partition since you have used up all of your filegroups created when you first built the partitioning scheme. Even though you are using the PRIMARY filegroup for all your partitions, you must still tell the partitioning scheme to reuse the PRIMARY filegroup for the new partition.

Figure 6 **Moving older data to archived tables**

```
--Prepare database
IF OBJECT_ID('active_data') IS NOT NULL
DROP TABLE active_data
GO

IF OBJECT_ID('archive_data') IS NOT NULL
DROP TABLE archive_data
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Active_Scheme')
DROP PARTITION SCHEME Active_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Active_Partition')
DROP PARTITION FUNCTION Active_Partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Archive_Scheme')
DROP PARTITION SCHEME Archive_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Archive_Partition')
DROP PARTITION FUNCTION Archive_Partition
GO

--Create active function
CREATE PARTITION FUNCTION Active_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Create archive function
CREATE PARTITION FUNCTION Archive_Partition (int) AS RANGE LEFT
FOR VALUES (100,200,300)


--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Active_Scheme
AS PARTITION Active_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Archive_Scheme
AS PARTITION Archive_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])


CREATE TABLE active_data
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
) ON Active_Scheme (col1)

CREATE TABLE archive_data
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
) ON Archive_Scheme (col1)

INSERT INTO active_data VALUES (1,'Description')
INSERT INTO active_data VALUES (2,'Description')
INSERT INTO active_data VALUES (3,'Description')
INSERT INTO active_data VALUES (4,'Description')
INSERT INTO active_data VALUES (10,'Description')
INSERT INTO active_data VALUES (11,'Description')
INSERT INTO active_data VALUES (12,'Description')
INSERT INTO active_data VALUES (13,'Description')
INSERT INTO active_data VALUES (14,'Description')
INSERT INTO active_data VALUES (100,'Description')


INSERT INTO archive_data VALUES (200,'Description')
INSERT INTO archive_data VALUES (300,'Description')
INSERT INTO archive_data VALUES (400,'Description')

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('active_data'),OBJECT_ID('archive_
      data'))

--Switch ownership of partition to another table
ALTER TABLE active_data SWITCH PARTITION 3 TO archive_data PARTITION 1

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('active_data'),OBJECT_ID('archive_
      data'))
```

Figure 7 **Moving and dropping data**

```
--Prepare database
IF OBJECT_ID('active_data') IS NOT NULL
DROP TABLE active_data
GO

IF OBJECT_ID('archive_data') IS NOT NULL
DROP TABLE archive_data
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Active_Scheme')
DROP PARTITION SCHEME Active_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Active_Partition')
DROP PARTITION FUNCTION Active_Partition
GO

--Create active function
CREATE PARTITION FUNCTION Active_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Active_Scheme
AS PARTITION Active_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

CREATE TABLE active_data
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
) ON Active_Scheme (col1)

CREATE TABLE archive_data
(
col1 INT PRIMARY KEY CLUSTERED
,col2 VARCHAR(15)
)

INSERT INTO active_data VALUES (1,'Description')
INSERT INTO active_data VALUES (2,'Description')
INSERT INTO active_data VALUES (3,'Description')
INSERT INTO active_data VALUES (4,'Description')
INSERT INTO active_data VALUES (10,'Description')
INSERT INTO active_data VALUES (11,'Description')
INSERT INTO active_data VALUES (12,'Description')
INSERT INTO active_data VALUES (13,'Description')
INSERT INTO active_data VALUES (14,'Description')
INSERT INTO active_data VALUES (100,'Description')

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('active_data'),OBJECT_ID('archive_
      data'))

--Switch ownership of partition to another table
ALTER TABLE active_data SWITCH PARTITION 3 TO archive_data

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('active_data'),OBJECT_ID('archive_
      data'))

--Drop all archive data without logging
DROP TABLE archive_data
GO

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] IN (OBJECT_ID('active_data'),OBJECT_ID('archive_
      data'))
```

Merge two partitions SQL Server allows you to drop single partitions from a table while keeping the data. This can be used to merge older active data into archived data or to reduce the number of partitions you have, easing your administration of the partitioned table. You can also use this to merge partitions, moving data from one filegroup to another to free up disk space on certain drive arrays. The code in **Figure 4** shows how you can move data from one partition into another partition on the same filegroup.

Move a single partition table into a partitioned table During load routines, large amounts of data often must be loaded into the database and then modified or aggregated before it is moved into the actual data table. SQL Server 2005 partitioning lets you move a single partition table into a table with multiple partitions. This means you can load data into a single load table, modify that data and then move the entire table into an existing table without the overhead of moving each individual row of data. This layer of partitioning does not involve altering the underlying partitioning structures. It involves modifying the partitioned table. The code in **Figure 5** shows how you can achieve this.

Move a partition from one table to another A common administrative task is to move older data into separate archive tables. The archiving process usually involves a series of statements that can create additional resource usage in your transaction logs. Switching the ownership of a partition from one table to another, however, is a simple way to archive large amounts of data without the transaction log overhead. This feature allows the database administrator to move segments of older data from their active tables to archived tables. But since the data is not actually moved, the amount of time it takes can be dramatically less than when moving individual rows of data. **Figure 6** shows how you can do this.

Use a single partition to create a new table You can move a single partition from an existing partitioned table into an empty non-partitioned table. By doing so, a database administrator can perform index maintenance on the single partition or easily drop large amounts of data without having that delete process logged. The sample in **Figure 7** shows how to move a partition into an empty table and then use that new table to drop the data.

Partitioned indexes

With the ability to partition the data of a table comes the ability to create partitioned indexes. This allows the database administrator to design the index structure based on the divided data rather than on the data of the entire table.

You can create indexes against tables where the data in the index is not aligned to the data in the table

Creating partitioned indexes results in individual B-trees on the partitioned indexes. The division of the indexes has the effect of creating smaller indexes that are more easily maintained by the storage engine during data modification, addition, and deletion. These smaller indexes can also be maintained individually by the database administrator, allowing for better index maintenance on large datasets.

Creating partitioned indexes When creating partitioned indexes, you can either create aligned indexes or non-aligned indexes. For aligned indexes, you create the index with a direct relationship to the partitioned data. (For non-aligned indexes, you choose a different partitioning schema.)

Aligned is the preferred method and will be done automatically if you create the partition table and then create the indexes without specifying a different partitioning scheme. Using aligned indexes gives you the flexibility to create additional partitions on the table and the ability to switch the ownership of a partition to another table. These capabilities are often the reason database administrators create partitioned tables in the first place, and simply using the partitioning schema of the table for your indexes will likely achieve your partitioning goals.

You can create indexes against tables where the data in the index is not aligned to the data in the table. If the data is in a partitioned table, this will allow you to join the data in different ways (partitioned data can be efficiently joined with other partitioned data by the query optimiser). Alternatively, you can do this with a non-partitioned table, allowing you to create a partitioned index (against the single partition table) so you can ease index maintenance.

The code in **Figure 8** will create a partitioned, non-clustered index on a partitioned table. The non-clustered index will be aligned with the table and will utilise the partitioning column of the table as the non-clustered index key.

The code in **Figure 9** will create a non-aligned, non-clustered index on a partitioned table. This non-clustered index will use different columns for its index key, which can be used in collated joins against other partitioned tables.

Maintain partitioned indexes In the past, performing index maintenance against large tables containing millions or even billions of rows of data often took longer than database administrators had time for. This maintenance was

Figure 8 **Partitioned, non-clustered index on partitioned table**

```
--Prepare database
IF OBJECT_ID('multiple_partition') IS NOT NULL
DROP TABLE multiple_partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Primary_Left_Scheme')
DROP PARTITION SCHEME Primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Left_Partition')
DROP PARTITION FUNCTION Left_Partition
GO


--Create partitioned table
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
    ALL TO ([PRIMARY])

CREATE TABLE multiple_partition
(
col1 INT
,col2 VARCHAR(15)
) ON Primary_Left_Scheme (col1)

--Create partitioned non-clustered index
CREATE NONCLUSTERED INDEX cl_multiple_partition ON multiple_
      partition(col1)

INSERT INTO multiple_partition VALUES (1,'Description')
INSERT INTO multiple_partition VALUES (2,'Description')
INSERT INTO multiple_partition VALUES (3,'Description')
INSERT INTO multiple_partition VALUES (4,'Description')
INSERT INTO multiple_partition VALUES (10,'Description')
INSERT INTO multiple_partition VALUES (11,'Description')
INSERT INTO multiple_partition VALUES (12,'Description')
INSERT INTO multiple_partition VALUES (13,'Description')
INSERT INTO multiple_partition VALUES (14,'Description')
INSERT INTO multiple_partition VALUES (100,'Description')
INSERT INTO multiple_partition VALUES (101,'Description')
INSERT INTO multiple_partition VALUES (102,'Description')
INSERT INTO multiple_partition VALUES (103,'Description')
INSERT INTO multiple_partition VALUES (104,'Description')

--Verify partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('multiple_partition')

--Verify index partitions
SELECT partition_id, index_id FROM sys.partitions pt
WHERE pt.[object_id] = OBJECT_ID('multiple_partition')
```

Figure 9 **Non-aligned, non-clustered index on partitioned table**

```
--Prepare database
IF OBJECT_ID('multiple_partition') IS NOT NULL
DROP TABLE multiple_partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Primary_Left_Scheme')
DROP PARTITION SCHEME Primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_schemes
WHERE [name] = 'Index_primary_Left_Scheme')
DROP PARTITION SCHEME Index_primary_Left_Scheme
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Left_Partition')
DROP PARTITION FUNCTION Left_Partition
GO

IF EXISTS(SELECT [name] FROM sys.partition_functions
WHERE [name] = 'Index_Left_Partition')
DROP PARTITION FUNCTION Index_Left_Partition
GO

--Create partitioned index function
CREATE PARTITION FUNCTION Index_Left_Partition (int) AS RANGE LEFT
FOR VALUES (10,50,100)

--Create partitioned table
CREATE PARTITION FUNCTION Left_Partition (int) AS RANGE LEFT
FOR VALUES (1,10,100)

--Place all index partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Index_primary_Left_Scheme
AS PARTITION Index_Left_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

--Place all partitions into the PRIMARY filegroup
CREATE PARTITION SCHEME Primary_Left_Scheme
AS PARTITION Left_Partition
--Partition must currently exist in database
      ALL TO ([PRIMARY])

CREATE TABLE multiple_partition
(
col1 INT
,col2 INT
) ON Primary_Left_Scheme (col1)

--Create non-aligned partitioned nonclustered index
CREATE NONCLUSTERED INDEX cl_multiple_partition ON multiple_
     partition(col2)
ON Index_primary_Left_Scheme (col2)

INSERT INTO multiple_partition VALUES (1,10)
INSERT INTO multiple_partition VALUES (2,10)
INSERT INTO multiple_partition VALUES (3,10)
INSERT INTO multiple_partition VALUES (4,10)
INSERT INTO multiple_partition VALUES (10,50)
INSERT INTO multiple_partition VALUES (11,50)
INSERT INTO multiple_partition VALUES (12,50)
INSERT INTO multiple_partition VALUES (13,50)
INSERT INTO multiple_partition VALUES (14,50)
INSERT INTO multiple_partition VALUES (100,100)
INSERT INTO multiple_partition VALUES (101,100)
INSERT INTO multiple_partition VALUES (102,100)
INSERT INTO multiple_partition VALUES (103,100)
INSERT INTO multiple_partition VALUES (104,100)

--Verify row count on partitioned data
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('multiple_partition')
AND p.index_id = 0

--Verify row count on partitioned index
--Row counts will not match those found in the data partitions
SELECT OBJECT_NAME(ps.[object_id])
,ps.partition_number
,ps.row_count
FROM sys.dm_db_partition_stats ps
INNER JOIN sys.partitions p
ON ps.partition_id = p.partition_id
AND p.[object_id] = OBJECT_ID('multiple_partition')
AND p.index_id <> 0
```

frequently left unperformed due to the data being locked while the index was being rebuilt. With SQL Server 2005, the database administrator can perform index maintenance online without locking the underlying table for a long period of time. But even this approach (which has you performing index maintenance while users are accessing the data) can still slow down your system due to resource usage. A better approach is to partition indexes into smaller segments and then perform index maintenance against those smaller partitions. For instance, to perform index maintenance against one index partition, you could simply append the code snippet below to the end of the code shown in **Figure 8**.

```
ALTER INDEX cl_multiple_partition
ON multiple_partition
REBUILD Partition = 2
```

Note that index maintenance against single index partitions must be performed offline and can cause locking of the table during the index maintenance. To prevent this, you can move the single partition into a separate partition, perform the index maintenance, and then move the partition back into the main table. This process will cause some performance issues as the partition is being moved back into the table and the clustered index is being updated, but this is less problematic than the locking of the entire table and requires fewer system resources.

Summary

As you can see, SQL Server 2005 table partitioning provides much improved flexibility for the storage and maintenance of data in large tables, without having to rework application code or SQL Server processes. With these abilities, SQL Server proves itself as a capable platform for enterprise-level, critical databases. ■

Noah Gomez *is a Senior SQL Server Development DBA for Verizon who specialises in VLDBs and large-scale applications. He is a member of the Professional Association for SQL Server (PASS) and was on the Verizon DBA team that worked on the multi-terabyte VLDBs that won Winter Corp Top Ten Grand Prize awards in 2003.*