# The Partitioned Table Framework (v3)

Author: Hans-Peter Portenlaenger  
<mailto:hapepo@hotmail.com>  
January, 2017

## Overview

When implementing table partitioning in Microsoft SQL Server, there is often the need to automate certain activities that recur on a regular basis, like archiving the oldest partition and switching in new partitions. Typically, these chores are being accomplished by SQL Agent Jobs that run on a scheduled basis, or by control flow tasks within SSIS packages. Implementing these tasks using T-SQL scripts turns out to be a cumbersome and error prone venture and often does not provide the flexibility to account for changes in the underlying database objects.

The set of stored procedures presented in this document provides an easy to use framework for painless development of these scripts, called the **Partitioned Table Framework** (PTF). The main focus of this framework is on the development of scripts for the so-called sliding-window scenario. However, there are also other scenarios in partition management where PTF eases the script development. Once you master their usage, the development cycles for the necessary scripts can be reduced from several days to just a few hours. You just have to pass in the required parameters, like table names and partition boundaries, and the procedures take care of all the steps to be performed in order to achieve the desired results. Since all the SQL statements being generated within the procedures are created on the fly, based on the current information in the system meta-data, all changes to the underlying database objects are taken into account at each procedure call. Thus, when adding a column to a table, altering an index, or adding a constraint after the scripts have been developed, your scripts can remain unchanged. Additionally, all the actions are executed in a transactional context. If something fails during their execution, everything is rolled back to the state before the procedure has been called.

Since all the operations performed by the framework are stateless, you may combine the PTF procedures in any order and apply any SQL command to the partitioning objects within the same script, without introducing undesired side-effects.

And, PTF is secure. All PTF procedures are executed **as caller**, and hence, a user cannot gain any elevated user rights just by calling into PTF. Simply put, a user just has the same rights he would have when not using PTF. Furthermore, all dynamically created statements are built such that no SQL injection is possible.

Version v3 of the framework is supported on all SQL Server versions starting from SQL Server 2008 up to SQL Server 2016. The main focus of v3 was to add support for new features in SQL Server 2016, like temporal tables or Always Encrypted columns. This is also the first version of PTF to support Azure SQL Database V12, which also allows for partitioned tables. With SP1 of SQL Server 2016 the supported footprint became even larger as starting with SP1 partitioned tables now can be created on Standard Edition and even Express Edition servers.

The framework must be installed (by running the install script “install\_ptf.sql”) on each database where partitioned tables should be managed by PTF scripts. To avoid name clashes, the objects of the framework are created within two separate database schemas, named **ptf** and **ptf\_internal**.

## Restrictions of PTF

The following restrictions are being enforced by PTF besides those already imposed by SQL Server:

* Although SQL Server allows non-partitioned indexes to exist on partitioned tables, these indexes would prevent partitions to be switched in and out. And, since PTF is all about partition switching, PTF does not support non-partitioned indexes on partitioned tables.
* An index on a partitioned table may be partitioned using a different partition scheme than the one the base table is partitioned with. But all partition schemes used for indexes must be based on the same partition function as the one used for the base table. The partition scheme by which the base table is being partitioned is called the primary partition scheme, whereas all other partition schemes that may be used by partitioned indexes or partition-aligned indexed views are called secondary partition schemes. Although the framework supports secondary partition schemes, some functions must be performed manually on secondary partition schemes.
* To split a partition in PTF it must be empty, i.e. it must not contain any data.
* PTF assumes that a partitioned table always has an empty partition either at the low end, for partitions defined by a RANGE RIGHT partition function, or at the high end, for partitions defined by a RANGE LEFT partition function (see below).

## Special partitions used by PTF

When PTF is used to implement a sliding-window scenario, there are a few special partitions with fixed designations that the user must be aware of:

* **Empty partition**. This is a partition on the left or right end, which must not contain any data. It’s a good idea to devise a special file group for this, that does not have any data file assigned to it. Hence, any attempt to insert data would trigger an error.
* **Next output partition**. This is the partition that contains the data to be switched out next.
* **Next input partition**. This is the partition, where data will be switched in next (and therefore must be empty).

Note, if you already have a partitioned table, but have missed to add the Empty partition, PTF has the procedure **ptf.AddEmptyPartition** up its sleeves doing just that for you without incurring any data movement. See the specification section for details.

Here’s the layout of these special partitions depending on the type of the underlying partition function:

**Partition:**

RANGE LEFT

1

2

n

n - 1

RANGE RIGHT

T

Next output partition

Next input partition

Empty  
partition

Empty partition

## The phases of partition switching

In a sliding-window scenario the work to be performed while switching partitions in and out usually can be separated into different phases:

Phase 1: First, the staging table to be switched into the next input partition must be created. In most cases, this table will not have any indexes to allow for fast data loading. But in some cases, it might be advantageous to have at least the clustered (columnstore) index in place before starting the data load. This task is being implemented in PTF by the procedure **CreateStagingTable**.

Phase 2: Next, the staging table must be filled with data. This usually is being accomplished through Bulk Insert or a SSIS data flow task and is out of scope for PTF.

Phase 3: Once all the data have been loaded into the staging table, this table must be prepared for the upcoming partition switch. There are quite a few requirements that must be met for a table to be switched in, e.g. all indexes must be identical to those of the partitioned table and foreign key and check constraints must match on both sides. In PTF, procedure **PrepareStagingTable** is responsible for accomplishing these preparation steps on the staging table.

Phase 4: In this phase the partition switches take place. This phase should be completed as fast as possible, since this is the phase where the partitioned table must be locked. The following activities are likely to be performed in this phase:

* the next output partition is switched out into an archive table,
* the switched out partition is being merged,
* if there is a check constraint in place on the partitioned table that enforces an upper bound for the values of the partitioning column, this upper bound must be adjusted accordingly before the staging table is being switching in,
* an additional check constraint must be created on the staging table that restricts the values of the partitioning column to the boundaries of the target partition,
* the staging table is switched into the next input partition and dropped afterwards,
* a new partition is being created by a split operation and turned into the new next input partition.

Within PTF, all you must do to make these steps happen is a single call to procedure **SwitchPartition**.

Phase 5: In the final phase the archive table being created in phase 4 is likely to be moved to a separate storage area to reclaim the space within its current file group. Another important step might be to update the statistics of the partitioned table after the data load. Besides this, the file group where the staging table has been switched in might get backed up. By designating this file group as READ ONLY, you also might exploit the partial backup feature of SQL Server. Since the requirements of this phase are diverse, but otherwise not difficult to script, in this phase there is no direct support provided by PTF, besides the fact that **SwitchPartition** has an OUTPUT parameter that returns the name of the file group that is mapped to the partition where the staging table has been switched in. This return value can be used, for instance, in a subsequent BACKUP FILEGROUP statement performed during this phase.

## The sliding window scenario

To demonstrate the ease of use of the PTF procedures, I’ll give a short example of how to apply PTF within the previously outlined phases. Let’s say, we have a partitioned fact table, named **dbo.Fact**, in our data warehouse holding the last 12 months of business data. Each month’s data are stored in a separate partition with a separate file group mapped to each partition. For each row of the fact table, the value of the column **create\_date** determines the partition this row belongs to, and consequently, **create\_date** represents our partitioning column. During the current month, data are loaded on a daily basis into the staging table, **dbo.Staging**. At the end of each month the partition containing the oldest data is being switched out into an archive table and the staging table is being switched into the last partition, thus implementing a so-called sliding-window scenario.

The partition function in our example is defined as:

CREATE PARTITION FUNCTION PFDate (datetime) AS RANGE RIGHT FOR VALUES (

'2015-07-01', '2015-08-01', '2015-09-01', '2015-10-01',

'2015-11-01', '2015-12-01', '2015-01-01', '2015-02-01',

'2016-03-01', '2016-04-01', '2016-05-01', '2016-06-01',

'2016-07-01')

And the 14 partitions spawned by this function are mapped to the file groups PRIMARY and FG01 through FG13 by:

CREATE PARTITION SCHEME PSFact AS PARTITION PFDate TO (

[PRIMARY], FG01, FG02, FG03, FG04, FG05, FG06,

FG07, FG08, FG09, FG10, FG11, FG12, FG13)

Note that since the partition function is of type RANGE RIGHT, the first partition always must be left empty and thus can be mapped to any file group, e.g. PRIMARY in our case.

And finally, the partitioned fact table looks like:

CREATE TABLE dbo.Fact (..., create\_date datetime NOT NULL, ...,

CONSTRAINT check\_date CHECK (create\_date < '2016-07-01')

) ON PSFact(create\_date)

Note that there is a check constraint in place that prevents data from being entered into the last partition, which represents the next input partition. There also may be a check constraint that prevents data to be entered into the first partition (the empty partition). But this is not as crucial as data that would flow into the last partition. Therefore, a check constraint on the low end is not directly supported by PTF and, if such a check constraint exists, you must manage it on your own.

Now let’s start with the discussion of the PTF procedures. As we have seen, in the first phase we have to create the staging table. In PTF we simply would call:

EXEC ptf.CreateStagingTable 'dbo.Fact', 'dbo.Staging'

which creates the staging table ‘Staging’ as an exact copy of the partitioned table ‘Fact’. Yet, the new table does not have any indexes defined on it. You might wonder why we do not have to specify the file group in which this table will be created. But the framework dictates that the staging table always will be switched into the next input partition, and therefore, the table by default is being created in the file group that is mapped to this partition, FG13 in our case.

Next, after the staging table has been filled with the data from July 2016, it’s time to switch this table into the partitioned table. But before doing so, a lot of modifications must be applied to the staging table in order to fulfill all the requirements that have to be met by a table before it can be switched into a partition. In PTF everything is being accomplished by calling:

EXEC ptf.PrepareStagingTable   
 'dbo.Fact', 'dbo.Staging', 'check\_date', 'page'

This procedure determines all the indexes that exist on the partitioned table and duplicates them on the staging table, if there’s an indexed view that references the partitioned table it creates an equivalent indexed view that references the staging table, and finally, it adds foreign key and check constraints to the staging table to match those of the partitioned table. Since all the data existing in the staging table would violate the current definition of the check constraint ‘check\_date’, this constraint must be exempted from those being created on the staging table, and therefore we specify the name of this constraint as the third parameter. And finally, we want the data to be PAGE compressed, which we specify in the fourth parameter.

Figure 1 shows the partition layout and its file group mappings after the staging table has been created and before the partition switch takes place.

PRIMARY

FG02

FG13

FG11

FG12

FG03

FG01

2

14

13

12

3

4

1

Staging

Partitions

Figure 1: Layout of the partitions, file groups and tables before the partition switch.

Now it’s time to perform the actual partition switches. The following steps must be performed:

* The partition with the oldest data, those from July 2015, must be switched out into an archive table (If the file group mapped to the output partition is read only, it’s being set to read/write before creating the archive table and performing the switch operation).
* The next output partition is being removed by a merge operation.
* The check constraint ‘check\_date’ must be adopted for the new data to be inserted.
* An additional check constraint for the partitioning column is added to the staging table to make sure that none of the values is outside the target partition’s boundaries.
* If the compression of the staging table differs from the target partition’s compression setting, the target partition is rebuilt accordingly.
* The prepared staging table is being switched into the target partition, which is the next input partition.
* The staging table is being dropped.
* The file group mapped to the input partition is set to read only.
* The last partition is being split to create a new last partition for the next month to come.

In PTF all these steps simply can be initiated by the following procedure call:

EXEC ptf.SwitchPartition

@part\_table = 'dbo.Fact',

@archive\_table = 'dbo.Archive',

@staging\_table = 'dbo.Staging',

@split\_value = '2016-08-01',

@check\_name = 'check\_date',

@make\_readonly = 1

The new partition added will be mapped to the file group that before had been mapped to the partition that was switched out, FG01 in our case. Therefore, the available file groups are being utilized in a round-robin fashion. Again, this demonstrates the beauty of PTF, i.e. you do not have to worry about file groups and partition numbers. Everything is seamlessly handled by PTF.

Figure 2 shows the new partition layout and its file group mappings after **SwitchPartition** has been executed:

FG01

FG13

FG12

FG04

FG03

FG02

PRIMARY

Partitions

Archive

4

3

2

1

12

13

14

Figure 2: Layout of the partitions, file groups and tables after the partition switch.

Now, the final step left would be to create backups and to move the ‘Archive’ table to some archive storage, e.g. a separate database, and thus reclaiming the data space before it is being reused.

## Incremental data load scenario

There’s another scenario where PTF can be helpful. Let’s say you want to load your data daily, but you want your partitions to hold one week of data. Of course, you could load your data each day directly into the target partition, but with an increasing number of indexes on the partitioned table or with stringent concurrency requirements, this technique might become prohibitive. In this situation, it might be worth considering incremental table switches, where each day you switch in a staging table that contains the current day plus all previous days of the week, i.e. on the first day you just load day one, on the second day you load day one and two, and so on. Although this technique requires nearly double the disk space during the load phase and requires data to be copied twice, using this technique the influence on the partitioned table is kept minimal (see also [Loading Bulk Data into a Partitioned Table](http://msdn.microsoft.com/en-us/library/cc966380.aspx)).

PTF supports this scenario by means of partition swapping. In this mode, the contents of the partition to be swapped is switched out into a temporary archive table, then the staging table is switched in, next this table is dropped, and finally the temporary archive table becomes the new staging table. Thus, after the swap the staging table contains the data from the swap partition prior to the swap, and the swap partition the data from the staging table before the swap.

Again, this is best demonstrated by code. In this example our partitioned table is again partitioned on a datetime column. Since this scenario is best supported using a RANGE LEFT partition function, the partition scheme of our table will be based on a RANGE LEFT partition function, where the boundaries are defined by an ordered set of weekend dates. The specifics of the partition function and scheme are omitted here.

Now let’s say, we want to enter the data for the week ending on July 10th, 2016 into one table partition. Here are the steps to be performed each day of the week:

1. On the first day of the week, we prepare our table’s partition function and scheme for the forthcoming week’s swap activity. Note, that the partition where the swap is being performed, always must be the partition preceding the last one. Therefore, we drop the oldest partition (or save it, if needed), and insert a new partition for this week’s data right before the last (empty) partition and map this partition to the file group used by the dropped partition:  
     
   EXEC ptf.SwitchPartition  
    @part\_table = 'dbo.Fact',  
    @archive\_mode = 'DROP',  
    @split\_value = '2016-07-10'
2. Next, the staging table is being created in the file group mapped to the partition added in the previous step:   
   EXEC ptf.CreateStagingTable 'dbo.Fact', 'dbo.Staging'
3. The current day’s data are being collected in a table, called dbo.Today.
4. Once the data load is complete, the data from dbo.Today are copied into our staging table.
5. Now, we can swap the partition by calling:  
   EXEC ptf.SwapPartition 'dbo.Fact', 'dbo.Staging'  
     
   Note, that we do not have to prepare the staging table before the swap. This is implicitly handled by the SwapPartition procedure.
6. Finally, the data from dbo.Today are copied again into our staging table and dbo.Today is being truncated/dropped afterwards.

On days 2 through 6 we repeat the steps 3, 4, 5 and 6.  
And on day 7, we repeat the steps 3 and 4 and then proceed with:

1. EXEC ptf.SwapPartition 'dbo.Fact', 'dbo.Staging' , @is\_last = 1  
     
   When specifying @is\_last = 1 the contents of the swap partition before the swap isn’t saved and the staging table is simply dropped.
2. Truncate/drop table dbo.Today.

Again, this is very easy to script, without worrying about file groups and their partition mappings. Also, note, that between the first call and last call to SwapPartition, the data in the swap partition must not be modified, or otherwise all modifications would be lost at the next swap step. In v3 during the first call to SwapPartition a trigger is being created on the partitioned table that prevents any modifications within the swap partition. This trigger is dropped again when SwapPartition is being called with @is\_last = 1.

## Partition update scenario

Another scenario where PTF is useful is when you must switch out a partition for a maintenance task that would be too costly to perform directly within the partitioned table or that would interfere with other activities on the partitioned table, e.g. performing a mass update on most of the records of a single partition.

The procedures in PTF that come in handy here are **SwitchOut** and **SwitchIn**. **SwitchOut** simply switches the specified partition out into the given switch-table, and **SwitchIn** switches the switch-table back into the partition it was before. The partition to switch either is identified by the partition number or by a value of the partitioning column that falls within the range of values of the partition to switch.

For example, if you wanted to perform updates on a table that has a nonclustered columnstore index (starting with SQL 2016 this index is updateable), you first have to drop this index and recreate it after the updates. On a partitioned table, it might be sufficient updating just the rows pertaining to a single partition. In such a case, one would switch out the partition that contains the rows to be updated. In PTF we would call:

EXEC ptf.SwitchOut 'dbo.Fact', 'dbo.TempFact', '2016-05-01'

This moves the data of the partition where ‘2016-05-01’ falls into its range out into the new table ‘TempFact’ together with all indexes defined on the partitioned table, except the nonclustered columnstore index. Thus, you can immediately start performing the desired updates. Once finished with the updates, you would call:

EXEC ptf.SwitchIn 'dbo.Fact', 'dbo.TempFact', '2016-05-01'

This time, before the switch back, the nonclustered columnstore index automatically is being recreated before performing the switch. Again, you just have to specify what to switch, but do not have to take care about all the organizational stuff involved with partition switching, like index creation and so on.

**SwitchOut** and **SwitchIn** also can be used as an alternative method for incremental partition loads. Just switch out the current partition when new data are to be loaded into this partition, insert the new data into the switch table, and switch this table back into the current partition. The major difference of this method and the method using **SwapPartition** is the time interval in which the current partition is not available. When using **SwapPartition** this time span is sub-second, whereas when using **SwitchOut** and **SwitchIn** the current partition is not available between the switch out and the switch in, hence, it mainly depends on the time it takes updating the data, and possibly recreating the indexes on the switch table. But this method does not have any additional storage requirements and is straight-forward to implement.

## Drop granularity differs from insert granularity

Let’s say you have a partitioned table, where the data from each day are stored in a separate partition. But, you want to purge old data only on a monthly basis. Hence, you add partitions daily, but drop them monthly. You therefore need to drop several partitions in one sweep. For this scenario, PTF offers the procedure **ptf.DropPartitions**, which allows for dropping/saving several consecutive partitions in single call. Simply specify at which partition to start the drop (by default, that’s the next output partition) and how many partitions you want to drop, or where to stop dropping.

If you have a table ‘Fact’ partitioned by day using a RANGE RIGHT partition function and you want to drop and save the partitions belonging to the oldest month, which happens to be June 2015, you simply would call  
  
EXEC ptf.DropPartitions 'dbo.Fact', 'dbo.Archive', @stop\_at = '2015-07-01'

This call drops the first 30 partitions for each day in June 2015 from table ‘Fact’ and moves the data from those partitions into a new partitioned archive table named ‘dbo.Archive’. You don’t have to worry about the partition layout of the table ‘Archive’, all is done implicitly. And, once you have saved the data from table ‘Archive’ and no longer need this table, you simply may call  
EXEC ptf.DropTable 'dbo.Archive'  
to drop the table together with the partition function and partition scheme created for it. Note, in v3 you can even truncate the data without copying them to an archive table.

## Converting a table into a partitioned table

Although the SQL Server Management Studio provisions a Create Partition Wizard that guides you through the process of converting an existing table into a partitioned table, you should not try this on any big sized table. Experienced users prefer to recreate the clustered index with the option DROP\_EXISTING = ON and specifying the partition scheme and thereby converting the table into a partitioned table. Although this process can convert even a huge table within a reasonable time frame, it’s still an awkward process. And, for instance, does not work for tables with a clustered columnstore index!

For your convenience, PTF provides an easy to use procedure for doing just that and even is slightly faster (about 20-30%, but your mileage may vary) than the method using the clustered index recreate. You just must pass in the name of the table you intend to partition, the partition scheme, and the partitioning column, and everything else is handled by the procedure logic. For instance

EXEC ptf.PartitionTable 'MyFact', 'MyScheme', 'create\_date'

partitions the table **MyFact** by the values in column **create\_date** using the partition scheme **MyScheme**. You may also compress the data along the way. Just add the desired compression mode as a parameter and the partitioned table will be created with your data being compressed.

## Installation

To install PTF, you simply have to execute the script **install\_ptf.sql** in each database where you want PTF to manage your partitioned tables. You must be a member of the **sysadmin** fixed server role or the **db\_owner** fixed database role to install or uninstall PTF. PTF cannot be installed in any of the system databases (except model). If you run the script within SQL Server Management Studio, make sure you have enabled SQLCMD Mode for the Query Window.

To uninstall PTF, you must run the script **uninstall\_ptf.sql** in the database where PTF previously has been installed. The uninstall script drops all objects that have been created within the database schemas **ptf** and **ptf\_internal** and finally drops these database schemas.

## Troubleshooting

PTF provides several features that might be helpful identifying the cause of a problem within PTF.

Each of the procedures in PTF has a parameter @debug, which, when set to 2, executes the procedure in debug mode, i.e. no actions are being performed and instead all the T-SQL statements that would have been issued by this procedure when passing the given set of parameters are being printed. Thus, you can check the statements to be executed beforehand and see if any error would be triggered. Note that error discovery is somewhat relaxed when running a procedure in debug mode, e.g. a staging table that does not exist, does not trigger an error in this mode. This allows for debugging a complete script where creating a staging table and working with the staging table occurs in the same script. But this also has the consequence that the statements being printed might not be exactly the same statements being issued in non-debug mode.

All the T-SQL statements that have been executed on behalf of the PTF procedures are being logged in the table **ptf.Log** with the date and time of execution and the SPID of the executing process. But note, since most of the statements are being executed within a database transaction, in case of a transaction rollback, also the information logged is being rolled back. Hence, you might not see the offending statement(s) in the Log table in case of an error. This can be avoided by running the procedure with the @debug parameter set to 1, since in this mode everything that is written into the Log table, also is being printed out (see also the below section on **PTF v2 changes**).

Also, the start and completion time of any of the high- procedures together with the passed in parameters is being logged in the Log table. The start event has a state of 0, and the completion event a state of 1 in the success case, or -1 in case of an error. Depending on the frequency of PTF usage, it is advisable to clear the log table from time to time using the procedure **ptf.ClearLog**.

For your convenience, PTF also includes the view **ptf.PartitionDetails** which is a join over the system catalog views **sys.partition\_functions**, **sys.partition\_schemes**, **sys.destination\_data\_spaces** and **sys.partition\_range\_values** which provides a quick view of the partitions that exist within a database and might be helpful to shed more light on a problem situation.

## PTF v2 changes

Besides some minor bug fixes, PTF v2 introduces a few new procedures and support for new features that have been implemented in SQL Server 2014. Besides this, the PTF objects have been reorganized, moving the internal procedures to a separate schema, called **ptf\_internal**.

Another important change is the use of an autonomous transaction when inserting rows into the Log table which allows the log records to be persisted even if the surrounding transaction is being rolled back. Since SQL Server does not support autonomous transactions, they are being emulated using a loopback linked server connection. During installation of PTF we try to create such a linked server, named **ptf\_loopback**, by calling the procedure **ptf.AddLoopbackLink**. Since creating a linked server requires special privileges, you might get an error during installation due to missing privileges, in which case you should ask a sysadmin user to execute **ptf.AddLoopbackLink** (without any parameters) in your DB. Note that there is just one linked server created per SQL Server instance, even if PTF is installed in several DB’s on your instance. Also, in case your DB is mirrored using DBM or AlwaysOn Availability Groups, you have to activate **ptf.AddLoopbackLink** manually on all secondaries. If the linked server is not present, logging does not fail, it simply works the way it did in v1.

The key feature in SQL Server 2014 supported by PTF v2, is the clustered columnstore index. For most procedures the support is seamless. But there are a few notable exceptions with special treatment for this new index type:

* In **ptf.CreateStagingTable** the default for parameter **@is\_clustered** becomes 1 for columnstore source tables, hence, the copy of a columnstore table is again a columnstore table by default.
* All procedures that have a **@compression** parameter, now also support the new compression mode specifiers COLUMNSTORE and COLUMNSTORE\_ARCHIVE.

Two more features in SQL Server 2014 which are addressing partitioned tables, are also supported by PTF v2. One is **Managed Lock Priority**. This feature makes partition switches more predictable and increases their concurrency. Whenever a partition switch takes place, a SCH-M lock must be acquired on the tables participating in the switch. If this lock cannot be granted due to a currently running query on one of the tables which holds a conflicting lock, the switch operation must wait until this query finishes. And, in data warehouse scenarios this may take some time. But, while the switch operation is waiting, no other queries on the tables run, even if they would not conflict with the currently running query, because they have to queue up behind the waiting switch operation. But with Managed Lock Priority the waiting switch operation can be a smart guy and step aside, waiting in a special low priority queue, and thereby allowing other queries to execute side by side with the blocking query and not having to wait. But this politeness has its limits. After a specified number of minutes, the waiting switch operation takes a specified action. Either it simply returns to the normal wait queue, or it terminates, or it terminates the blocking queries. In PTF there are two parameters that can be passed to all procedures that may perform partition switches: **@low\_prio\_wait** defines the maximum number of minutes to wait in the low priority queue, and **@abort\_action** allows to specify the action to take if the SCH-M lock still could not be granted after the wait time elapses, and can be either ‘NONE’, ‘SELF’, or ‘BLOCKERS’ according to the aforementioned actions. Passing NULL to **@abort\_action** reverts to the behavior prior to SQL Server 2014.

So far, PTF did not deal with updating the statistics on the partitioned table after partitions have been switched in or out. It was the caller’s responsibility to perform the update on the statistics any time after such a data change has happened. Now, SQL Server 2014 supports **Incremental Statistics** on partitioned tables that allow for explicitly recomputing statistics on individual partitions and automatically merging the partition level statistics into a table statistic. Support for Incremental Statistics is seamless in PTF v2. Whenever a procedure that performs a partition switch detects indexes and statistics that were enabled for incremental statistics, it will perform an update of the statistics on the modified partitions. Therefore, in order to make this happen, simply enable all indexes and statistics on your partitioned tables for incremental statistics by running a **UPDATE STATISTICS <tab name> WITH NORECOMPUTE, INCREMENTAL = ON** and everything else is handled by PTF.

Without any doubt, memory-optimized OLTP is the hottest new feature in SQL Server 2014. Although memory optimized tables cannot be partitioned, PTF v2 allows memory optimized tables in certain scenarios, e.g. **ptf.CreateStagingTable** can be used to convert a memory optimized table into a disk based table.

But there are also a few changes in v2 which are not related to new SQL features. The procedure **ptf.DropPartition** is being replaced by the new procedure **ptf.DropPartitions**. The main enhancement of the new procedure is the ability to drop several consecutive partitions in a single call. To allow for archiving of the dropped partitions, in case of more than one partition being dropped, the archive table to be created has to be partitioned as well. The new procedure handles the automatic creation of a partition function and scheme for this purpose. This new procedure comes in handy in situations where partitions are being added to a table on a, let’s say, daily schedule. But you want the dropping of aged-out partitions to happen with a coarser granularity, let’s say, weekly or monthly. In that case, you can use **ptf.DropPartitions** to move out all partitions pertaining to the oldest week or month in a single step.

Complementary to this procedure, there’s another new utility procedure, **ptf.DropTable**, which can be used to drop in a single call the partitioned archive table created by **ptf.DropPartitions** together with the partition function and scheme that has been created for the archive table.

And finally, **ptf.AddEmptyPartition** adds an empty partition to the partition range, in case this has been omitted when creating the partition layout, without incurring any data movement. PTF v3 changes

## PTF v3 changes

Now, v3 of PTF can be installed on Azure SQL Databases V12 which also supports partitioned tables. Since PTF is a solution fully contained within a database, it is suited very well for a PaaS implementation like Azure SQL Database. The only difference to an on-premise SQL Server instance is that the only file group available is PRIMARY, and it cannot be set to read-only. Also, creating a loopback linked server is not possible and hence no autonomous transaction for logging is possible.

V3 also supports most of the relevant new features of SQL Server 2016. All tables used also can be temporal tables (where applicable) and using Always Encrypted columns is transparent to PTF. All new index combinations regarding columnstore indexes are supported, as well as filtered nonclustered columnstore indexes. The only feature that is not yet supported are stretch tables.

For all table name parameters one can also specify the name of a synonym. And for most parameters that did expect the name of a partitioned table, also a non-partitioned table can be specified if it makes sense. E.g. you can use procedure SwitchOut to switch a non-partitioned table into another non-partitioned table. And now, tables may also have a FILESTREAM column and may be partitioned by the corresponding ROWGUID column.

In v3 the procedure **ptf.PartitionTable** has been redesigned, now allowing for in-place transformation into a partitioned table. It also supports the use of a fast mode that allows transformation without data movement in special situations. See the specification for more details on that.

## PTF specification

The procedures of the framework are organized in two different levels. The low-level procedures provide the basic functionality and are applicable on a wide range of usage scenarios. The high-level procedures are just wrappers around the low-level procedures, and while having a few more restrictions on their usage, they are much easier to use than their low level counterparts and usually are your first choice when implementing your partitioning scripts.

## High-level procedures

Creates a new staging table as a non-partitioned copy of the source table, optionally together with a clustered index.

**Syntax**

|  |
| --- |
| CreateStagingTable [ @source\_table = ] ’*table\_name*’ , [ @staging\_table = ] ’*staging\_table\_name*’ [ , [ @fg\_name = ] { ’*filegroup\_name’* | ’DEFAULT’ } [ OUTPUT ] ] [ , [ @is\_clustered = ] *cluster\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_table** = ] '*table\_name*'

Is the name of a table in the current database that acts as the source table. *table\_name* is **nvarchar(256)**, with no default.

[ **@staging\_table** = ] '*staging\_table\_name*'

Is the name of the staging table to be created. If no schema name is given within *staging\_table\_name*, the table will be created in the default schema of the user executing the stored procedure. An object with that name must not yet exist within this database schema. *staging\_table\_name* is **nvarchar(256)**, with no default.

[ **@fg\_name** = ] { '*filegroup\_name*' | ‘default’ } [ OUTPUT ]

Is the name of the file group in which the staging table will be created. *filegroup\_name* is **sysname**, with a default of NULL. If a name has been specified, it must be the name of an existing file group in the current database, or the string ‘DEFAULT’, in which case the table will be created in the default file group of the database. If no file group name has been specified, the table will be created in the file group that currently is mapped to the next input partition, if the source table is partitioned, or otherwise, in the file group where the source table is located in. After the call, the name of the file group in which the table has been created will be returned in this parameter.

[ **@is\_clustered** = ] *cluster\_flag*

Is 0 or 1, with a default of 0 for row-store tables and 1 for column-store tables. If 1 is specified and the source table is clustered, an equivalent clustered index will be created for the staging table.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

Usually, the new table created does not have an index (except for a FILESTREAM table, which always has a key constraint on the ROWGUID column). If you want the new table to be clustered the same way the source table is clustered, set the parameter *@is\_clustered* to 1. If the source table has a clustered columnstore index, *@is\_clustered* has the value 1 by default.

The new table will be a non-partitioned one. If the source table is also not partitioned, the new table will be created in the same file group as the source table. But, if the source table is partitioned, the table will be created in the file group that is mapped to the current next input partition (therefore, at least two partitions must exist, as otherwise no next input partition is defined). One can always override the default file group designation by specifying a file group name in *@fg\_name* (in this case, no clustered index can be created).

Default constraints that exist on source table columns also will be defined identically for the corresponding columns in the new table. On SQL Server 2016 and above, if the source table is a temporal table, the period for system\_time setting will be preserved. The same is true for Always Encrypted columns.

**Permissions**

Requires REFERENCES permission on the source table, CREATE TABLE permission on the database and ALTER permission on the target schema.

**Examples**

The following example shows how to create the new staging table ‘Staging\_Fact’ based on the source table ‘dbo.Fact’. If the source table is partitioned, the table will be created in the file group that is mapped to the next input partition, or otherwise in the same file group as the source table. If the source table is clustered, the staging table will be clustered as well.

|  |
| --- |
| EXEC ptf.CreateStagingTable @part\_table = ’dbo.Fact’,  @staging\_table = ’dbo.Staging\_Fact’,  @is\_clustered = 1; |

**See also**

CreateTableClone

Prepares a staging table before it can be switched into the next input partition of a partitioned table.

**Syntax**

|  |
| --- |
| PrepareStagingTable [ @part\_table = ] ’*table\_name*’ , [ @staging\_table = ] ’*staging\_table\_name*’ [ , [ @check\_name = ] ’*constraint\_name*’ ] [ , [ @compression = ] ’*compression\_mode*’ ] [ , [ @compress\_all = ] *all\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] '*table\_name*'

Is the name of a table in the current database. *table\_name* is **nvarchar(256)**, with no default. All indexes and constraints that exist on this table will be created identically on the staging table.

[ **@staging\_table** = ] '*staging\_table\_name*'

Is the name of the staging table to be prepared. *staging\_table\_name* is **nvarchar(256)**, with no default. This table must exist in the current database. The only index that may exist on this table is a clustered index, and if the table is a FILESTREAM table, the key constraint on the ROWGUID column.

[ **@check\_name** = ] '*constraint\_name*'

Is the name of a check constraint defined on a column (typically the partitioning column) of the source table. This check constraint will be omitted from those being created on the staging table. Such a check constraint must be specified, if creating the constraint on the staging table fails due to some data in the staging table violating the associated check condition. *constraint\_name* is **sysname**, with a default of NULL.

[ **@compression** = ] ‘*compression\_mode’*

Defines the compression mode to be applied to the staging table or its clustered index and possibly all the other indexes. Valid values are NULL, ‘NONE’, ‘ROW’, ‘PAGE’, ‘COLUMNSTORE’ or ‘COLUMNSTORE\_ARCHIVE’, where NULL is equivalent to ‘NONE’. ‘NONE’, ‘ROW’ and ‘PAGE’ only apply to b-tree indexes, whereas, ‘COLUMNSTORE’ and ‘COLUMNSTORE\_ARCHIVE’ only applies to column-store indexes. *compression\_mode* is **nvarchar(20)**, with a default of NULL.

[ **@compress\_all** = ] *all\_flag*

Is 0 or 1, with a default of 0. If 0 is specified, the compression mode will only be applied to the staging table or its clustered index, otherwise all indexes and indexed views will be compressed accordingly.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

For a table to be switched into a partition of a partitioned table or to a non-partitioned table, the tables must meet all requirements for this switch to succeed. Like, the indexes on both tables must exist identically and be located in the same file groups, matching check and foreign key constraints must exist for both tables, if the target table is referenced by an indexes view, so must the staging table, and so on. This procedure takes care of all these requirements and adds the missing objects to the staging table.

It’s possible for the staging table to exist in a different file group than the one mapped to the destination partition (e.g. if a file group name was specified in the **CreateStagingTable** call). In this case, the staging table will be moved to the correct destination file group while creating the clustered index. This can be beneficial when performing compression since the table will be compressed during the move which saves space in the destination file group and minimizes page fragmentation.

**Permissions**

REFERENCES permission on the partitioned table and ALTER permission on the staging table.

**Examples**

The following example prepares the staging table ‘Staging\_Fact’ after it has been filled with data and before the table is going to be switched into the next input partition of the partitioned table ‘dbo.Fact’. The partitioned table’s check constraint ‘check\_date’ shall not be duplicated on the staging table since the data in ‘Staging\_Fact’ would violate this constraint. The staging table and all its indexes will be page-compressed.

|  |
| --- |
| EXEC ptf.PrepareStagingTable @part\_table = ’dbo.Fact’,  @staging\_table = ’Staging\_Fact’,  @check\_name = ’check\_date’,   @compression = ’page’,  @compress\_all = 1; |

**See also**CreateStagingTable  
CreateIndexesAndConstraints

Completes the necessary steps to perform the sliding window algorithm on a partitioned table.

**Syntax**

|  |
| --- |
| SwitchPartition [ @part\_table = ] ’*partitioned\_table\_name’*  [ , [ @archive\_table = ] ’*archive\_table\_name*’ ] [ , [ @staging\_table = ] ’*staging\_table\_name*’ ] [ , [ @split\_value = ] *boundary\_value* ] [ , [ @new\_filegroup = ] ’*filegroup\_name*’ ] [ , [ @check\_name = ] ’*constraint\_name*’ ] [ , [ @upper\_bound = ] *boundary\_value* ][ , [ @make\_readonly = ] *readonly\_flag* ] [ , [ @low\_prio\_wait = ] *wait\_time* ] [ , [ @abort\_action = ] *’abort\_action’* ] [ , [ @archive\_mode = ] *’archive\_action’* ] [ , [ @fg\_name = ] *filegroup\_variable* OUTPUT ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*partitioned\_table\_name’*

Is the name of the partitioned table in the current database where the partition switch will take place. *partitioned\_table\_name* is **nvarchar(256)**, with no default.

[ **@archive\_table** = ] '*archive\_table\_name*'

Is the name of an archive table into which the next output partition of the partitioned table will be switched to, if *archive\_action* is ‘SAVE’ or NULL. This table must not yet exist. *archive\_table\_name* is **nvarchar(256)**, with a default of NULL.

[ **@staging\_table** = ] '*staging\_table\_name*'

Is the name of the staging table to be switched into the next input partition of the partitioned table. *staging\_table\_name* is **nvarchar(256)**, with a default of NULL. If *staging\_table\_name* is NULL, no table will be switched in.

[ **@split\_value** = ] *boundary\_value*

If a *boundary\_value* is specified, the last partition will be split at this boundary, thus creating a new partition at the right end of the partition range. Hence, the specified value must be greater than any boundary value currently defined for the partition function. *boundary\_value* is **sql\_variant** with a base type compatible with the type of the partitioning column, with a default of NULL. If the partition is of type RANGE RIGHT and a staging table name has been specified, a split value also must be specified.

[ **@new\_filegroup** = ] '*filegroup\_name*'

Is the name of the file group to be mapped to the new partition created by the partition split. *filegroup\_name* is **sysname**, with a default of NULL.

[ **@check\_name** = ] '*constraint\_name*'

If *constraint\_name* is specified, a column level check constraint with that name will be created for the partitioning column on the partitioned table. If a check constraint with that name already exists, that constraint will be dropped prior to creating the new one. This constraint typically is used to define an upper limit for the values in the partitioning column. *constraint\_name* is **sysname**, with a default of NULL.

[ **@upper\_bound**  = ] *boundary\_value*

Is the upper bound to be used in the constraint’s check condition. *boundary\_value* is **sql\_variant** with a base type compatible with the type of the partitioning column, with a default of NULL. An upper bound must be specified, if no staging table name has been given or the partition is of type RANGE RIGHT, and no split value has been specified.

[ **@make\_readonly** = ] *readonly\_flag*

Is 0 or 1, with a default of 0. If 1 is specified, the file group the next input partition is mapped to is set to READ ONLY.

[ **@low\_prio\_wait** = ] *wait\_time*

The number of minutes to wait in the low priority wait queue if there is a conflicting lock on the partitioned table when performing the partition switch. *wait\_time* is **int** with a default of 0.

[ **@abort action** = ] *’abort\_action’*

Specifies the action to take if there is still a conflicting lock after *wait\_time* has been elapsed. *abort\_action* is nvarchar(8), with a default of NULL, which means that no transfer into the low priority wait queue takes place. This is the only possible action on SQL Server versions below 2014. The other actions are NONE, i.e. we remain in the normal wait queue, SELF, i.e. the waiting session terminates, and BLOCKERS means that the blocking session will be terminated (which requires the caller to have the ALTER ANY CONNECTION privilege.

[ **@archive\_mode** = ] *’archive\_action’*

Specifies the action to be performed on the next output partition. *archive\_action* is **nvarchar(10)**, with a default of NULL. The other possible values are ‘NONE’, ‘SAVE’ and ‘DROP’, where ‘NONE’ means do nothing, ‘SAVE’ switches the partition out to the specified archive table, and ‘DROP’ removes all rows from the partition. With the latter two options, the next output partition is going to be removed by means of a partition merge operation. Using NULL is equivalent to using ‘NONE’ when *archive\_table\_name* is NULL, or ‘SAVE’ otherwise.

[ **@fg\_name** = ] *filegroup\_variable* **OUTPUT**

If *staging\_table\_name* has been specified, upon completion, the file group of the staging table will be returned in this variable. *filegroup\_variable* is an output variable of type **sysname**.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure manages the various tasks to be performed around table switching in a sliding window scenario, like moving partition data in and out, dropping and creating partitions, or managing a check constraint that enforces an upper bound for the partitioning column. Each task can be activated separately, or all of them together in a single procedure call. In the following, each of the tasks is outlined separately:

The next output partition may be removed and the data in the partition may be saved or dropped. If an archive table name has been specified and @archive\_mode is NULL or ‘SAVE’ the data is being switched out to this table prior to the partition merge. This table must not yet exist since it will be created as an exact copy of the partitioned table together with a clustered index, if the partitioned table is also clustered. Alternatively, the data in the next output partition simply may be dropped, if @archive\_mode is set to ‘DROP’. If any of the file groups mapped to this output partition currently is set to READ ONLY, it will be changed to READ WRITE prior to saving or dropping the partition.

If the name of a staging table is specified, this table is going to be switched into the next input partition of the partitioned table. This target partition must be empty for the switch to succeed and the staging table must be fully prepared (see **PrepareStagingTable**) prior to the switch. Before the switch, a column level check constraint is created on the staging table that restricts the values of the partitioning column to the boundaries of the target partition. If the staging table or one of its indexes is compressed, the corresponding target partition will be compressed accordingly prior to the switch. Thus, the target partition adopts the compression settings of the staging table. Hence, if you want the target partition to be compressed, the staging table must be compressed accordingly during the prepare stage.

A new partition may be created at the right end of the partition range by performing a split operation. This is being accomplished by specifying the value used for the split. This split value must be greater than all existing boundary values, and hence, only the last partition can be split. Besides the split value, the procedure also must know the file group the new partition shall be mapped to. This can be accomplished in several ways:

First, the name of the file group the new partition should be mapped to can be specified as an additional parameter. But this only works for the table’s primary partition scheme. If there are also secondary partition schemes, the file groups for those must be specified beforehand using explicit ALTER PARTITION SCHEME NEXT USED statements. Of course, you also may use such a statement for the primary partition scheme and omit the file group name parameter.

But, if the next output partition has been removed within this procedure call (**@archive\_mode** is ‘SAVE’ or ‘DROP’), it would be reasonable to utilize the file group mappings of this dropped partition for the new partition. Thus, if the next output partition has been removed and no file group name was specified, the new partition simply will adopt the file group mappings from the dropped next output partition.

Otherwise, the file group the current next input partition is mapped to will be used. Hence, the new partition inherits the mapping from its predecessor.

If the name of a check constraint is being specified, a column-level check constraint will be created on the partitioned table that defines an upper bound for the values of the partitioning column. If a check constraint with that name already exists, it will be dropped beforehand. The upper bound being used can be directly specified as a parameter. But in case a new partition is also being created, the upper bound likely will be identical to the split value. Therefore, if a split value has been specified, the upper bound parameter may be omitted and the split value will be used instead for the upper bound in the check constraint. The actual check condition used depends on the type of the underlying partition function:  
for a RANGE LEFT partition function the condition will be “*partitioning column* **<=** *upper bound*”  
and for a RANGE RIGHT partition function it will be “*partitioning column* **<** *upper bound*”.

Next, the file group mapped to the next input partition can be set to READ ONLY.

And finally, if a staging table has been switched in, the staging table, now being empty, will be dropped.

All operations performed by this procedure are meta-data only operations (except for the check constraint being created) and thus should complete within seconds, unless the operations are being blocked by existing locks on the partitioned table. In this case, the switch request can be put into the low priority wait queue for a specified amount of time, and if the lock still could not be granted after the wait time elapses, the switch request either can be put back on the normal wait queue, or terminate itself, or terminate the blocking session. This functionality is only available starting with SQL Server 2014.

If a filegroup must be set to read/write and the operation cannot be performed because other users still have an open session within the database, the procedure fails. Whereas a change of a filegroup to read only that cannot be performed is simply logged in table ptf.Log.

**Permissions**

CONTROL or ALTER and ALTER ANY DATASPACE permission on the database is required.

**Examples**

The following example shows how to perform a sliding window operation on the partitioned table ‘dbo.Fact’, which is partitioned on a datetime column using a RANGE RIGHT partition function defining partitions each holding data for one month.

In this example, we want to switch in the already prepared staging table ‘Staging\_Fact’ containing the June 2016 data. At the same time, we want to switch out the partition containing the oldest data into the archive table ‘Archive\_Fact’. The switched-out partition will be merged and a new partition will be added for the next month to come, i.e. July 2016. No file group is being specified for the new partition. Hence, the new partition will be mapped to the file group(s) that had been mapped to the merged next output partition.

Also, a check constraint with name ‘check\_date’ will be dropped and then recreated on table ‘Fact’ using @split\_value as the new upper bound. Upon completion, the file group mapped to the next input partition will be marked as READ\_ONLY and the staging table will be dropped. Additionally, the file group mapped to the next input partition will be returned in the variable @filegroup and may be used in a subsequent BACKUP FILEGROUP statement.

|  |
| --- |
| DECLARE @filegroup sysname;  EXEC ptf.SwitchPartition @part\_table = ’dbo.Fact’,   @archive\_table = ’Archive\_Fact’,  @archive\_mode = ’SAVE’,  @staging\_table = ’Staging\_Fact’,   @split\_value = ’2016-07-01’,  @check\_name = ’check\_date’,  @make\_readonly = 1,  @fg\_name = @filegroup OUTPUT; |

**See also**

PrepareStagingTable  
SwitchPartition2

Performs the swap of a single partition within a partitioned table.

**Syntax**

|  |
| --- |
| SwapPartition [ @part\_table = ] ’*table\_name’,*  [ @staging\_table = ] ’*staging\_table\_name*’ [ , [ @*is\_last* = ] *last\_flag* ] [ , [ @make\_readonly = ] *readonly\_flag* ] [ , [ @compression = ] ’*compression\_mode*’ ] [ , [ @compress\_all = ] *all\_flag* ] [ , [ @low\_prio\_wait = ] *wait\_time* ] [ , [ @abort\_action = ] *’abort\_action’* ] [ , [ @fg\_name = ] *filegroup\_variable* OUTPUT ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*table\_name’*

Is the name of a table in the current database where the partition swap will take place. The table must not be a temporal history table. *table\_name* is **nvarchar(256)**, with no default.

[ **@staging\_table** = ] '*staging\_table\_name*'

Is the name of the staging table to be swapped in. This table must not yet be prepared, i.e. it may only have a clustered (columnstore) index. *staging\_table\_name* is **nvarchar(256)**, with no default.

[ **@is\_last** = ] *last\_flag*

Is 0 or 1, with a default of 0. If *last\_flag* is 1, this is the last call in a sequence of partition swaps. In this case, the staging table will be dropped after swapping, otherwise the staging table will contain all rows from the swap partition prior to the swap, but no indexes.

[ **@make\_readonly** = ] *readonly\_flag*

Is 0 or 1, with a default of 0. If *readonly\_flag* is set to 1, the file group mapped to the swap partition will be changed to read only after the swap did take place. *readonly\_flag* can only be set when *last\_flag* is 1. Also, do not use in Azure SQL Database.

[ **@compression** = ] ‘*compression\_mode’*

Defines the compression mode to be applied to the staging table and possibly its indexes prior to the swap. Valid values are NULL, ‘NONE’, ‘ROW’, ‘PAGE’, ‘COLUMNSTORE’ or ‘COLUMNSTORE\_ARCHIVE’, where NULL is equivalent to ‘NONE’. ‘NONE’, ‘ROW’ and ‘PAGE’ only apply to b-tree indexes, whereas, ‘COLUMNSTORE’ and ‘COLUMNSTORE\_ARCHIVE’ only apply to column-store indexes. *compression\_mode* is **nvarchar(20)**, with a default of NULL.

[ **@compress\_all** = ] *all\_flag*

Is 0 or 1, with a default of 0. If 0 is specified, the compression mode will only be applied to the staging table or its clustered (columnstore) index, otherwise all indexes and indexed views will be compressed accordingly.

[ **@low\_prio\_wait** = ] *wait\_time*

The number of minutes to wait in the low priority wait queue if there is a conflicting lock on the partitioned table when performing the partition switch. *wait\_time* is **int** with a default of 0.

[ **@abort action** = ] *’abort\_action’*

Specifies the action to take if there is still a conflicting lock after *wait\_time* has been elapsed. *abort\_action* is nvarchar(8), with a default of NULL, which means that no transfer into the low priority wait queue takes place. This is the only possible action on SQL Server versions below 2014. The other actions are NONE, i.e. we remain in the normal wait queue, SELF, i.e. the waiting session terminates, and BLOCKERS means that the blocking session will be terminated (which requires the caller to have the ALTER ANY CONNECTION privilege.

[ **@fg\_name** = ] *filegroup\_variable* **OUTPUT**

The file group mapped to the swap partition will be returned in this variable. *filegroup\_variable* is an output variable of type **sysname**.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

Swapping a partition comprises the following steps: the swap partition is being switched out into a temporary archive table, next the staging table is being prepared and then switched into the now empty swap partition, then the staging table is dropped, and if *last\_flag* is 0, the temporary archive table will be renamed to the name of the staging table, i.e. the archive table becomes the new staging table. This new staging table does not have any indexes, except for a clustered columnstore index which will be preserved.

The swap partition is always the partition preceding the last one. Thus, if the table’s partition is of type RANGE RIGHT, where the next input partition defaults to the last partition, you first must add a new partition (e.g. by calling **SwitchPartition**) before performing the swap. Note, that in this case you should call **CreateStagingTable** before adding the new partition to create the staging table in the correct file group. For RANGE LEFT partitions, this is not necessary, since the next input partition is already the partition preceding the last one.

By the nature of the swap operation, all data modifications performed in the swap partition since the preceding swap, will be overridden by the next swap. Therefore, no data should be modified in the swap partition until the last swap has finished (specified with @is\_last = 1). In v3 this is being enforced by creating a trigger on the partitioned table before initiating the first swap, which rolls back any data modification on the swap partition. With the last swap, this trigger finally gets deleted.

**Permissions**

To run this procedure with **@make\_readonly** set to 1 CONTROL or ALTER permission on the database is required. All other modes require ALTER permission on the tables.

**Examples**

The following example shows how to perform a partition swap on the partitioned table ‘dbo.Fact’. After the call, the swap table ‘Staging\_Fact’ contains the contents of the swap partition prior to the call, and the swap partition has the contents of the swap table before the call.

|  |
| --- |
| EXEC ptf.SwapPartition @part\_table = ’dbo.Fact’,   @staging\_table = ’Staging\_Fact’; |

In the next example, we demonstrate the correct sequence of calls starting the swap for a table partitioned by a RANGE RIGHT function. First, the staging table is being created in the file group mapped to the next input partition, which always is the last partition for RANGE RIGHT. Next, a new partition is added by performing a partition split. Either we specify the file group this new partition is being mapped to, or we exploit the round-robin assignment of file groups by dropping the next output partition at the same time (as shown in the example). Finally, we perform the first swap on the partition preceding the last partition.

|  |
| --- |
| EXEC ptf.CreateStagingTable @part\_table = ’dbo.Fact’,  @staging\_table = ’dbo.Staging\_Fact\_1606’;  EXEC ptf.SwitchPartition @part\_table = ’dbo.Fact’,   @archive\_mode = ’DROP’,  @split\_value = ’2016-07-01’;  EXEC ptf.SwapPartition @part\_table = ’dbo.Fact’,   @staging\_table = ’dbo.Staging\_Fact\_1606’; |

**See also**

CreateStagingTable  
SwitchPartition  
SwitchPartition2

Switches a partition of a table out into a switch table

**Syntax**

|  |
| --- |
| SwitchOut [ @part\_table = ] ’*table\_name’,*  [ @switch\_table = ] ’*switch\_table\_name*’ [ , [ @*part\_value* = ] *partitioning\_value* ] [ , [ @part\_num = ] *partition\_number* ] [ , [ @keep\_indexes = ] *keep*\_*index\_specification* ] [ , [ @low\_prio\_wait = ] *wait\_time* ] [ , [ @abort\_action = ] *’abort\_action’* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*table\_name’*

Is the name of a table in the current database where the partition switch will take place. *table\_name* is **nvarchar(256)**, with no default.

[ **@switch\_table** = ] '*switch\_table\_name*'

Is the name of the table where the partition will be switched to. This table must not yet exist. *switch\_table\_name* is **nvarchar(256)**, with no default.

[ **@part\_value** = ] *partitioning\_value*

Is the value that determines the partition that will be switched out. *partitioning\_value* is of type **sql\_variant** with a default of NULL, and its base type must be compatible with the type of the partitioning column.

[ **@part\_num** = ] *partition\_number*

Is the number of the partition to be switch out. If the source table is partitioned, either *partitioning\_value* or a *partition\_number* must be specified, if both are specified *partition\_number* takes precedence and *partitioning\_value* is being ignored. *partition\_number* is **int** with a default of NULL.

[ **@keep\_indexes** = ] *keep\_index\_speicification*

*keep\_index\_specification* is **nvarchar(12)** with possible values ‘NONE’, ‘CLUSTERED’, ‘NONCLUSTERED’ and ‘ALL’, with ‘NONCLUSTERED’ being the default. If ‘NONE’ is specified, no index will be preserved on the switched out table, i.e. it will become a heap table. If ‘CLUSTERED’ is specified, only the clustered index will be kept if the source table is clustered as well. With ‘NONCLUSTERED’ the switched out table will be fully prepared with all indexes and constraints with the exception of a nonclustered columnstore index, and finally ‘ALL’ also keeps a nonclustered columnstore index if one exists on the partitioned table.

[ **@low\_prio\_wait** = ] *wait\_time*

The number of minutes to wait in the low priority wait queue if there is a conflicting lock on the partitioned table when performing the partition switch. *wait\_time* is **int** with a default of 0.

[ **@abort action** = ] *’abort\_action’*

Specifies the action to take if there is still a conflicting lock after *wait\_time* has been elapsed. *abort\_action* is nvarchar(8), with a default of NULL, which means that no transfer into the low priority wait queue takes place. This is the only possible action on SQL Server versions below 2014. The other actions are NONE, i.e. we remain in the normal wait queue, SELF, i.e. the waiting session terminates, and BLOCKERS means that the blocking session will be terminated (which requires the caller to have the ALTER ANY CONNECTION privilege.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure can be used if a partition must be switched out for performing a maintenance task on the partition’s data and the task is too costly to be performed directly on the whole table. Later the switch table can be switched back into the source table using the procedure **SwitchIn**.

If the source table is partitioned, the partition to be switched out is being determined either by specifying the partition number or by specifying a value that falls within the interval associated with the partition to be switched out. If the file group mapped to this partition is marked as READ\_ONLY, it will be changed to READ\_WRITE prior to the switch operation.

**Permissions**

ALTER permission on the partitioned table and ALTER permission on the target schema.

**Examples**

The following example shows how to switch out the partition of the table ‘dbo.Fact’ that contains the value ‘2016-07-01’ into table ‘Facts\_1607’, and cluster the new table the same way as the source table, but does not create any other index or constraint that exists on the source table.

|  |
| --- |
| EXEC ptf.SwitchOut @part\_table = ’dbo.Fact’,   @switch\_table = ’Facts\_1607’,  @part\_value = ’2016-07-01’,  @keep\_indexes = N’CLUSTERED’; |

**See also**SwitchIn

Switches a table back into a partition of a table.

**Syntax**

|  |
| --- |
| SwitchIn [ @part\_table = ] ’*table\_name’,*  [ @switch\_table = ] ’*switch\_table\_name*’ [ , [ @*part\_value* = ] *partitioning\_value* ] [ , [ @part\_num = ] *partition\_number* ] [ , [ @create\_indexes = ] *create\_index\_specification* ] [ , [ @make\_readonly = ] *readonly\_flag* ] [ , [ @low\_prio\_wait = ] *wait\_time* ] [ , [ @abort\_action = ] *’abort\_action’* ] [ , [ @fg\_name = ] *filegroup\_variable* OUTPUT ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*table\_name’*

Is the name of a table in the current database where the partition switch will take place. *table\_name* is **nvarchar(256)**, with no default.

[ **@switch\_table** = ] '*table\_name*'

Is the name of the table to be switched into the target partition. *switch\_table\_name* is **nvarchar(256)**, with no default.

[ **@part\_value** = ] *partitioning\_value*

Is the value that determines the target partition where the table will be switched in. *partitioning\_value* is of type **sql\_variant** with a default of NULL, and its base type must be compatible with the type of the partitioning column. If also *partition\_number* is NULL, the *partitioning\_value* is determined by retrieving from the switch table the first non-NULL value for the partitioning column.

[ **@part\_num** = ] *partition\_number*

Is the number of the target partition. *partition\_number* is **int** with a default of NULL. If both *partition\_number* and *partitioning\_value* are being specified, *partition\_number* takes precedence and *partitionig\_value* is being ignored.

[ **@create\_indexes** = ] *create\_index\_specification*

*create\_index\_specification* is **nvarchar(12)** with possible values ‘NONE’, ‘ALL’ and ‘COLUMNSTORE’, where ‘COLUMNSTORE’ is the default. If ‘ALL’ is specified, all the indexes and constraints being present on the source table also will be created on the switch table prior to the switch. If ‘NONE’ is specified, the table is already fully prepared, which is also the case with ‘COLMNSTORE’, except for a nonclustered columnstore index that might exist on the source table and therefore must be created on the switch table before the partition switch can take place. This flag usually works in concert with the **@keep\_indexes** parameter of the procedure SwitchOut. See the Remarks section for details.

[ **@make\_readonly** = ] *readonly\_flag*

Is 0 or 1, with a default of 0. If *readonly\_flag* is set to 1, the file group mapped to the switch partition will be changed to read only after the switch did take place. Do not use on Azure SQL Database.

[ **@low\_prio\_wait** = ] *wait\_time*

The number of minutes to wait in the low priority wait queue if there is a conflicting lock on the partitioned table when performing the partition switch. *wait\_time* is **int** with a default of 0.

[ **@abort action** = ] *’abort\_action’*

Specifies the action to take if there is still a conflicting lock after *wait\_time* has been elapsed. *abort\_action* is nvarchar(8), with a default of NULL, which means that no transfer into the low priority wait queue takes place. This is the only possible action on SQL Server versions below 2014. The other actions are NONE, i.e. we remain in the normal wait queue, SELF, i.e. the waiting session terminates, and BLOCKERS means that the blocking session will be terminated (which requires the caller to have the ALTER ANY CONNECTION privilege.

[ **@fg\_name** = ] *filegroup\_variable* **OUTPUT**

The file group mapped to the switch partition will be returned in this variable. *filegroup\_variable* is an output variable of type **sysname**.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure can be used when a partition has been switched out using procedure **SwitchOut**, and now should be switched back into the partitioned table.

If the target table is partitioned, the target partition either will be determined by specifying the number of the target partition, or by providing a value for the partitioning column, either explicitly in the **@part\_value** parameter, or implicitly by using the first non-NULL value for the partitioning column stored in the switch table. The target partition must be empty before the switch.

Here’s the list of matching parameter values for **@keep\_indexes** in the **SwitchOut** call and **@create\_indexes** in the corresponding **SwitchIn** call:

|  |  |
| --- | --- |
| **@keep\_indexes in SwitchOut** | **@create\_indexes in SwitchIn** |
| NONE | ALL |
| CLUSTERED | ALL |
| NONCLUSTERED | COLUMNSTORE |
| ALL | NONE |

After being switched in, the switch table will be dropped.

**Permissions**

ALTER permission on the partitioned table and CONTROL or ALTER permission on the switch table.

**Examples**

The following example shows how to switch back the table ‘Facts\_1607’ into the partition that contains the data of July 2016 within the partitioned table ‘dbo.Fact’.

|  |
| --- |
| EXEC ptf.SwitchIn @part\_table = ’dbo.Fact’,   @switch\_table = ’Facts\_1607’,  @part\_value = ’2016-07-01’; |

**See also**

SwitchPartition2  
SwitchOut

Deletes one or more partitions from a partitioned table and possibly saves the data of the dropped partitions in an archive table.

**Syntax**

|  |
| --- |
| DropPartitions [ @part\_table = ] ’*partitioned\_table\_name’*  [ , [ @archive\_table = ] ’*archive\_table\_name’* ] [ , [ @partition\_function = ] *’partition\_function\_name’* ] [ , [ @partition\_scheme = ] ’*partition\_scheme\_name’* ] [ , [ @start\_partition = ] *partition\_number* ] [ , [ @num\_partitions = ] *partition\_count* ] [ , [ @stop\_at = ] *boundary\_value* ] [ , [ @low\_prio\_wait = ] *wait\_time* ] [ , [ @abort\_action = ] *’abort\_action’* ] [ , [ @fg\_name = ] *filegroup\_variable* OUTPUT ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*partitioned\_table\_name’*

Is the name of a partitioned table in the current database where the partitions will be dropped from. *partitioned\_table\_name* is **nvarchar(256)**, with no default.

[ **@archive\_table** = ] ‘*archive\_table\_name’*

Is the name of the archive table to be created in the current database where the data of the dropped partitions will be switched to. An object of that name must not yet exist in the current database. *archive\_table\_name* is **nvarchar(256)**, with a default of NULL. If *archive\_table\_name* is NULL, the data in the dropped partitions will be discarded.

[ **@partition\_function** = ] ‘*partition\_function\_name’*

If more than one partition is being dropped, the archive table must be created as a partitioned table. If no *partition\_function\_name* has been specified, a partition function with a system generated name will be generated, otherwise, a partition function with the given name will be created for this purpose. *partition\_function\_name* is **sysname**, with a default of NULL.

[ **@partition\_scheme** = ] ‘*partition\_scheme\_name’*

If more than one partition is being dropped, the archive table must be created as a partitioned table. If no *partition\_scheme\_name* has been specified, a partition scheme with a system generated name will be generated, otherwise, a partition scheme with the given name will be created for this purpose. *partition\_scheme\_name* is **sysname**, with a default of NULL.

[ **@start\_partition** = ] *partition\_number*

Is the number of the first partition to be dropped. *partition\_number* is of type **int,** with the number of the **next output partition** being the default.

[ **@num\_partitions** = ] *partition\_count*

Is the number of consecutive partitions to be dropped starting from the start partition. *partition\_count* is **int,** with a default of 1. A value of -1 drops all partitions to the right of the start partition except the empty partition at the right end, if the partition is of type RANGE LEFT.

[ **@stop\_at** = ] *boundary\_value*

Instead of specifying the number of partitions to be dropped, one can also specify a valid boundary value that demarks the point where to stop deleting partitions. All partitions from the start partition up to the partition preceding the boundary value will be dropped. *boundary\_value* is of type **sql\_variant** with a default of NULL, and its base type must be compatible with the type of the partitioning column. The partition number of the partition identified by *boundary\_value* must be greater than or equal to the start partition number.

[ **@low\_prio\_wait** = ] *wait\_time*

The number of minutes to wait in the low priority wait queue if there is a conflicting lock on the partitioned table when performing the partition switch. *wait\_time* is **int** with a default of 0.

[ **@abort action** = ] *’abort\_action’*

Specifies the action to take if there is still a conflicting lock after *wait\_time* has been elapsed. *abort\_action* is nvarchar(8), with a default of NULL, which means that no transfer into the low priority wait queue takes place. This is the only possible action on SQL Server versions below 2014. The other actions are NONE, i.e. we remain in the normal wait queue, SELF, i.e. the waiting session terminates, and BLOCKERS means that the blocking session will be terminated (which requires the caller to have the ALTER ANY CONNECTION privilege.

[ **@fg\_name** = ] *filegroup\_variabl*e OUTPUT

Here one can specify a variable that upon return will contain the name of the file group where the archive table was created in, if just one partition has been dropped, or if more than one partition has been dropped, and all dropped partitions were mapped to the same file group, the name of this file group. You could use this information in a sliding window scenario, where file groups are assigned in a round robin fashion.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure removes the *partition\_count* partitions right of the start partition by performing a series of partition merges. If an archive table has been specified, the partitions to be dropped will be switched into corresponding partitions of the archive table prior to this. Instead of specifying the number of partitions to be deleted, one can also specify the *boundary\_value* where deletion stops, i.e. only partitions left of the *boundary\_value* will be dropped.

If more than one partition is being dropped, the archive table has to be partitioned. In this case a partition function and partition scheme must be created by the procedure for all the partitions to be dropped plus an additional empty partition that will be mapped to the same file group as the empty partition of the source table. Hence, the partitioned archive table being created is also conformant with the rules for partitioned tables in PTF. You can either specify a name for the partition scheme and function to be created, or otherwise system generated names will be used.

The **empty partition** (at the beginning or end of the partition range, depending on the type of the partition function) cannot be dropped. Therefore, *partition\_number* must be less than the number of partitions on the source table, i.e. at least one partition, the empty partition, must remain in the source table.

If any of the file groups mapped to the partitions to be dropped is set to READ ONLY, they will be changed to READ WRITE ahead of dropping the partitions.

**Permissions**

ALTER permission on the partitioned table, ALTER permission on the archive table’s schema, ALTER ANY DATASPACE permission, and ALTER permission on the database, if a partition function and scheme has to be created.

**Examples**

In the following example ‘dbo.Facts’ is a table that is partitioned by day using a RANGE RIGHT partition function. Each day of a 12-month period is kept in a separate partition. All partitions belonging to the same month are mapped to the same file group. At the end of each month, the oldest month’s data is being removed from the table and saved in the archive table ‘dbo.Old\_Facts’. Currently, the oldest data in the table are those from May 2015, and hence we stop dropping partitions at June 1st 2015. The information about the file group these oldest partitions have been mapped to is being used in a subsequent switch in of the staging table containing the data of the last day of the current month and creating a new partition for the first day of the next month.

|  |
| --- |
| DECLARE @next\_fg sysname;  EXEC ptf.DropPartitions @part\_table = ’dbo.Facts’,  @archive\_table = ’dbo.Old\_Facts’,  @stop\_at = ’2015-06-01’,  @fg\_name = @next\_fg OUTPUT;  EXEC ptf.SwitchPartition @part\_table = ’dbo.Facts’,   @staging\_table = ’dbo.Facts\_20160531’,   @split\_value = ’2016-06-01’,  @new\_filegroup = @next\_fg; |

**See also**

SwitchPartition

Drops a table in the current database.

**Syntax**

|  |
| --- |
| DropTable [ @table\_name = ] ’*table\_name’* [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@table\_name** = ] ‘*table\_name’*

Is the name of the table to be dropped. *table\_name* is **nvarchar(256)**, with no default.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure is most useful in situations where the partitioned archive table created during a call to procedure ptf.DropPartitions needs to be deleted after the data have been saved in some other location. Since a partition function and scheme have been created for the archive table, those are also deleted by this call, if no other object exists that is partitioned by this function and scheme.

The table to be deleted must not be referenced by a foreign key constraint or referenced by a schema bound function or view.

**Permissions**

ALTER permission on the table’s schema or ALTER permission on the database, if also the partition scheme and function are being deleted.

**Examples**

In the following example the oldest seven partitions (1 partition for each day of the week) are being dropped from table ‘dbo.Facts’ and moved into the partitioned archive table ‘dbo.Old\_Week\_Facts’. Next, the saved data are copied into a table within an archive database, and finally, to reclaim the space in the current database, the archive table is being dropped, together with the partition function and scheme associated with the archive table.

|  |
| --- |
| EXEC ptf.DropPartitions @part\_table = ’dbo.Facts’,  @archive\_table = ’dbo.Old\_Week\_Facts’,  @num\_partitions = 7;  INSERT INTO archive\_DB.dbo.Facts SELECT \* FROM dbo.Old\_Week\_Facts;  EXEC ptf.DropTable @table\_name = ’dbo.Old\_Week\_Facts’; |

**See also**

DropPartitions

Converts a non-partitioned table into a partitioned table.

**Syntax**

|  |
| --- |
| PartitionTable [ @source\_table = ] ’*source\_table\_name’,*  [ @partition\_scheme = ] ’*partition\_scheme\_name’*, [ @partition\_column = ] ’*column\_name*’ [ , [ @fg\_name = ] ’*filegroup\_name*’ ] [ , [ @compression = ] ’*compression\_mode*’ ] [ , [ @compress\_all = ] *all\_flag* ] [ , [ @incremental\_stats = ] *stats\_flag* ] [ , [ @fast\_mode = ] *fast\_mode\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_table** = ] ‘*source\_table\_name’*

Is the name of a non-partitioned table in the current database. *source\_table\_name* is **nvarchar(256)**, with no default.

[ **@partition\_scheme** = ] ‘*partition\_scheme\_name’*

Is the name of an existing partition scheme that is used to partition the source table. *partition\_scheme\_name* is of type **sysname** with no default.

[ **@partition\_column** = ] *’column\_name’*

Is the name of a column of the source table that will be used to partition the table. *column\_name* is **sysname** with no default. The data type of this column must be compatible with the type of the partition function the specified partition scheme is based on.

[ **@fg\_name** = ] '*filegroup\_name*'

Except when using fast mode, all data rows of the source table pertaining to a partition of the target table are first extracted into a temporary table which is then prepared and switched into its destination partition. By default, each of the temporary tables is created in the file group that is mapped to the corresponding target partition. In some cases, it might be beneficial to first create the temporary table in a separate staging file group and move the data to the destination file group during the prepare step. *filegroup\_name* is the name of the file group used as this staging area. *filegroup\_name* is **sysname**, with a default of NULL.

[ **@compression** = ] ‘*compression\_mode’*

Defines the compression mode to be applied to all partitions of the new partitioned table and possibly all its indexes. Valid values are NULL, ‘NONE’, ‘ROW’, ‘PAGE’, ‘COLUMNSTORE’ or ‘COLUMNSTORE\_ARCHIVE’, where NULL means that the current compression setting of the source table and its indexes will be preserved. *compression\_mode* is **nvarchar(20)**, with a default of NULL.

[ **@compress\_all** = ] *all\_flag*

Is 0 or 1, with a default of 0. If 0 is specified, the compression mode will only be applied to the partitioned table and/or its clustered index, otherwise also the nonclustered indexes will be compressed accordingly.

[ **@incremental\_stats** = ] *stats \_flag*

Is 0 or 1, with a default of 1. If 1 is specified, on SQL Server 2014 and above, the unfiltered indexes and statistics on the new partitioned table will be created with the NORECOMPUTE and INCREMENTAL options set to ON.

[ **@fast\_mode** = ] *fast\_mode\_ \_flag*

Is 0 or 1, with a default of NULL. If *fast\_mode\_flag* is 0, the data copy method will be used, if it is 1 and fast mode is not possible, an error will be returned that specifies the cause (in the state field), and if NULL, fast mode will be used, if possible, and data copy method otherwise. The different modes will be further explained in the Remarks section.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

The table to be partitioned must be a non-partitioned disk based user table that is not being replicated, not change tracked or used in CDC. The table must not have a full-text, XML or spatial index. The table must not be referenced by a FOREIGN KEY constraint and no indexed view may exist that references this table. All unique indexes on that table must contain the designated partitioning column as a key column. And, the type of the given partitioning column must match the type of the partitioning function the partition scheme is based upon.

If these requirements are met, an in-table transformation is being performed, i.e. first, the table is switched out into a temporary table, then, the now empty table is going to be partitioned using the specified partition scheme and the given partitioning column.

Next, there are two distinct modes how data is moved back from the temporary table to the now partitioned table. If @fast\_mode is 0 or @fast\_mode is NULL and the conditions for fast mode are not met, for each partition a staging table is created and the pertaining data copied from the temporary table to the staging table by means of an INSERT INTO SELECT command, and then, the staging table is prepared and switched into its target partition.

On the other hand, if the conditions for fast mode are met, and @fast\_mode is 1 or NULL, the temporary table is simply switched back into its target partition, which is partition 1 for a RANGE LEFT partition or for RANGE RIGHT partition that consist of just 1 partition, otherwise, its partition 2.

These are the conditions that must be met for fast mode to be applicable:

* All the values of the partitioning column are within the boundaries of the target partition.
* No compression takes place, i.e. @compression must be NULL.
* The table and all indexes exist in the file group the target partition is mapped to.
* If the table has a clustered index, the partitioning column is a key column of this index.
* If no clustered index exists, the partitioning column is part of each nonclustered index, either as a key, or as a non-key column.

Actually, fast mode only consists of partition switches and meta-data changes and therefore is very fast. All changes are performed within a transaction and usually can be performed while the table is being accessed by readers and writers. But, if you do not care if the existing data are further partitioned, and only want future data insertions to flow into separate partitions, that’s the preferred way to go.

In all other cases, partitioning the table is a time consuming and resource intensive process, consisting of data movements and index creations. During that time, the data in the table cannot be accessed (actually the table is empty). The final switch operations also run in a transaction. But the actual copy operations do not. But they are still reversible in case of an error. This has the additional benefit that you can truncate the log from time to time while the procedure is running in order to avoid huge transaction log growth.

In any case, it is advisable to create a database backup before commencing the operation.

**Permissions**

ALTER permission on the table, the table’s schema and CREATE TABLE permission on the database.

**Examples**

The following example shows how to convert the table ‘dbo.Fact’ into an equivalent partitioned table. The table and all its indexes will be partitioned by column ‘create\_date’ using the existing partition scheme ‘PS\_Fact’. Additionally, all indexes will be page compressed. The conversion is not using fast mode.

|  |
| --- |
| EXEC ptf.PartitionTable @part\_table = ’dbo.Fact’,  @partition\_scheme = ’PS\_Fact’,  @partition\_column = ’create\_date’,  @compression = ’PAGE’,  @compress\_all = 1,  @fast\_mode = 0; |

**See also**

CreateTableCloneAdds an empty partition at the beginning or the end of a partitioned table if none exists.

**Syntax**

|  |
| --- |
| AddEmptyPartition [ @part\_table = ] ’*partitioned\_table\_name’*, [ @split\_value = ] *boundary\_value* [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@part\_table** = ] ‘*partitioned\_table\_name’*

Is the name of a partitioned table in the current database where the empty partition shall be added. *partitioned\_table\_name* is **nvarchar(256)**, with no default.

[ **@split\_value** = ] *boundary\_value*

The empty partition is being added by inserting a new boundary value to the partition function the table is partitioned with. If the partition function is of type RANGE RIGHT, the new boundary value must be smaller than all existing boundary values and equal or less than the smallest value of the partitioning column in the table. Otherwise, if the partition function is of type RANGE LEFT, the new boundary value must be greater than all existing boundary values and equal or greater than the highest value of the partitioning column in the table. *boundary\_value* is **sql\_variant** with a base type compatible with the type of the partitioning column, with no default.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

One of the most predominant mistakes when starting to work with partitioned tables is the oversight to provision an empty partition at the low end (for RANGE RIGHT partitions) or the high end (for RANGE LEFT partitions) of the partition range. But once you start implementing your sliding-window scenario, sooner or later you will feel the pain when a partition merge or split takes longer as expected and comes along with high IO activity. The reason is that the neighboring partition completely must be copied into a new partition. But even if you recognize your fault before experiencing it the hard way, there’s no simple method to correct the fault. You must switch out the first/last partition, split the partition and switch the partition back. The purpose of this procedure is to implement these steps for you.

The inserted empty partition will be mapped to the same file group as the partition that was split.

**Permissions**

CONTROL or ALTER, CREATE TABLE and ALTER ANY DATASPACE permission on the database and ALTER permission on the partitioned table is required.

**Examples**

In the following example the partitioned fact table ‘dbo.Fact’ has been partitioned by a partition function of type RANGE RIGHT. So far, several monthly partitions have been created to hold data for the year 2015. Now, before adding data for January 2016, you want to switch out the January 2015 data and merge the first partition. But, since you did not cater for an empty partition at the left end of your partition range, the January 2015 data resides in partition 1. In this situation you could call **ptf.AddEmptyPartition** to add an empty partition right before the current partition 1. Afterwards you can use **ptf.SwitchPartition** to switch out the data from January 2015.

|  |
| --- |
| EXEC ptf.AddEmptyPartition @part\_table = ’dbo.Fact’,   @split\_value = ’2015-01-01’; |

**See Also**

SwitchPartition

## Low-level procedures

Creates a clone of a table.

**Syntax**

|  |
| --- |
| CreateTableClone [ @source\_tab\_id = ] *source\_table\_id* , [ @target\_table = ] ’*table\_name*’ [ , [ @fg\_name = ] { ’*filegroup\_name*’ | ‘default’ } [ OUTPUT ] ] [ , [ @part\_num = ] *partition\_number* ] [ , [ @ps\_id = ] *partition\_scheme\_id* ] [ , [ @switch\_source = ] *switch\_source\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_tab\_id** = ] *source\_table\_id*

Is the object ID of a table in the current database that acts as the source for creating the target table. *source\_table\_id* is **int**, with no default.

[ **@target\_table** = ] '*table\_name*'

Is the name of the table to be created. If no schema name is given within *staging\_table\_name*, the table will be created in the default schema of the user executing the stored procedure. An object with that name must not yet exist within this database schema. *table\_name* is **nvarchar(256)**, with no default.

[ **@fg\_name** = ] { '*filegroup\_name*' | ‘default’ } [ OUTPUT ]

Is the name of a file group in which the staging table will be created. *filegroup\_name* is **sysname**, with a default of NULL. If a name has been specified, it must be the name of an existing file group in the current database, or the string ‘default’, in which case the table will be created in the default file group of the database. The file group must not be read-only. After the call, the name of the file group in which the table was created will be returned in this parameter.

[ **@part\_num** = ] *partition\_number*

If the source table is partitioned or if **@ps\_id** is specified, this is the number of the partition which determines the file group the new table will be created in. *partition\_number* is **int**, with a default of NULL. *partition\_number* must be a valid partition number within the partition scheme, or 1 or NULL for a non-partitioned table.

[ **@ps\_id** = ] *partition\_scheme\_id*

Is the ID of a partition scheme in the current database. *partition\_scheme\_id* is **int**, with a default of NULL. If the source table is not partitioned, but the table’s file group shall be determined by a partition mapping, this parameter specifies the partition scheme to be used and **@part\_num** the partition used for the mapping.

[ **@switch\_source** = ] *switch\_source\_flag*

Is 0 or 1, with a default of 0. If 1 is specified the table created in the future is going to be switched into a partition of a partitioned table, and hence, if the source table is a temporal table, the new table also will have corresponding time period columns defined. This parameter is only relevant if the source table is a temporal table.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

The source table may be partitioned or not, and even can be a memory-optimized table. But the new table generated always will be a non-partitioned disk-based table. The new table will be created as an exact copy of the source table identified by *source\_table\_id*, but without constraints, except DEFAULT constraints, or the primary key or unique constraint defined on the rowguid column, if the table contains FILESTREAM data. Also, neither the IDENTITY nor the NOT FOR REPLICATION column properties are being preserved. The source table must not be a FileTable. If *switch\_source\_flag* is set and the source table is a temporal table, the row start and row end properties also will be preserved. For Always Encrypted columns, the encryption properties will be preserved as well.

If a *filegroup\_name* has been specified, the table will be created in this given file group. Otherwise, if a *partition\_number* has been specified and the source table is partitioned or a *partition\_scheme\_id* has been specified, the file group, where the table will be created in is being determined by the partition scheme mapping for the given *partition\_number*. Otherwise, the table will be created in the same file group as the source table or the default file group, if the source table is memory-optimized. This is also true for TEXTIMAGE data and non-partitioned FILESTREAM data. But, if the destination file group is determined by a partition mapping, a partition scheme must exist that maps to FILESTREAM file groups and that is based on the same partition function as the partition scheme for table data. The corresponding mapping is defined through *partition\_number*.

**Permissions**

Requires REFERENCES permission on the source table and CREATE TABLE permission on the database.

**Examples**

The following example shows how to create a table clone with the name ‘Staging\_Fact’ based on the partitioned source table ‘Fact’ in the file group that currently is mapped to partition 10 in the pertaining partition scheme.

|  |
| --- |
| DECLARE @objid int;  SET @objid = OBJECT\_ID(’dbo.Fact’);  EXEC ptf.CreateTableClone @source\_tab\_id = @objid,  @target\_table = ’dbo.Staging\_Fact’,  @part\_num = 10; |

**See also**

Creates an index on a table or a view that duplicates an existing index on a table or view.

**Syntax**

|  |
| --- |
| CreateIndex [ @source\_tab\_id = ] *source\_object\_id* , [ @index\_id = ] *index\_id* [ , [ @target\_object = ] ’*object\_name*’ ] [ , [ @constraint\_name = ] ’*constraint\_name*’ ] [ , [ @part\_num = ] *partition\_number* ] [ , [ @ps\_id = ] partition\_scheme\_id ] [ , [ @part\_scheme = ] ’*partition\_scheme\_name’* ] [ , [ @part\_col\_id = ] *partition\_column\_id* ] [ , [ @compression = ] *compression\_mode* ] [ , [ @incremental\_stats = ] *stats\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_tab\_id** = ] *source\_object\_id*

Is the object ID of a table or a view. *source\_object\_id* is **int**, with no default.

[ **@index\_id** = ] *index\_id*

Is the ID of an existing index on the table or view specified by *source\_object\_id*. *index\_id* is **int**, with no default. Neither XML indexes nor spatial indexes are being supported. If *index\_id* is 0, or *index\_id* is 1 and the corresponding clustered index already exists on the target table, no index will be created, but the table or clustered index will be rebuilt with the corresponding compression being applied, or if it’s a clustered columnstore index, the open delta rowgroup(s) will be closed and compressed.

[ **@target\_object** = ] '*object\_name*'

Is the name of an existing table or a view on which the index will be created the same way as on the source table/view. *object\_name* is **nvarchar(256)**, with a default of NULL. If no target object has been specified, the target is the source table, in which case the existing index on the source table is going to be partitioned using the DROP EXISTING = ON option.

[ **@constraint\_name** = ] '*constraint\_name*'

If the specified index represents a primary key or unique constraint, an explicit constraint name can be specified using this parameter. *constraint\_name* is **sysname**, with a default of NULL. If NULL is specified, the new constraint will be created with a system generated name. A database object with that name must not yet exist.

[ **@part\_num** = ] *partition\_number*

If the source table is partitioned or if **@ps\_id** is specified, this is the number of the partition which determines the file group the index will be created in. *partition\_number* is **int**, with a default of NULL. *partition\_number* must be a valid partition number within the partition scheme, or 1 or NULL for a non-partitioned table.

[ **@ps\_id** = ] *partition\_scheme\_id*

Is the ID of a partition scheme in the current database. *partition\_scheme\_id* is **int**, with a default of NULL. If the source table is not partitioned, but the index’s file group shall be determined by a partition mapping, this parameter specifies the partition scheme to be used and **@part\_num** the partition used for the mapping.

[ **@part\_scheme** = ] ‘*partition\_scheme\_name’*

Is the name of an existing partition scheme. This scheme will be used to partition the index or table (if *index\_id* is 0 or 1). *partition\_scheme\_name* is **sysname**, with a default of NULL.

[ **@part\_col\_id** = ] *partition\_column\_id*

If a *partition\_scheme\_name* has been specified, this is the ID of the column that will be used to partition the table/index. The data type of this column must be compatible with the type of the partition function the specified partition scheme is based on. *partition\_column\_id* is **int**, with a default of NULL.

[ **@compression** = ] *compression\_mode*

Defines the compression to be applied to the index or table (if *index\_id* is 0). *compression\_mode* is **int**, with a default of NULL. Valid values are 0, 1, 2, 3, 4 or NULL. Specifying NULL causes the index being compressed the same way as the corresponding partition of the source index. The values 0-4 correspond to the possible compression values in sys.partitions.

[ **@incremental\_stats** = ] *stats\_flag*

Is 0 or 1, with a default of 0. If 1 is specified, and the target table is partitioned and the index is not a filtered one and not a columnstore index, the corresponding index on the target table will be created with incremental statistics turned on. On SQL Server 2012 and below this parameter is being ignored.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

*index\_id* can refer to any clustered, nonclustered or columnstore index defined on the source object. XML and spatial indexes are not supported. This procedure creates an identical index on the target object. If the source table is partitioned or **@ps\_id** was specified, the partitioning column may be added if it is not yet part of the index.

If a *partition\_number* has been specified and the source table is partitioned or **@ps\_id** has been specified, the file group, where the index will be created in is being determined by the partition scheme mapping for the given *partition\_number*. Else, if a *partition\_scheme\_name* together with a *partition\_column\_id* has been specified, the index will be partitioned using this partition scheme and partitioning column.

Primary key or unique constraints will be created with a system defined name, unless a name was specified in **@constraint\_name**. All other indexes will be created with the same name as the source index.

If a target object has been specified, the specified index must not yet exist on the target object except for a clustered index, which may already exist on the target table. If the source object is a heap table, 0 may also be specified as the *index\_id*. In this case, no index is being created, but if a *compression\_mode* is being specified, the target table will be compressed accordingly.

**Permissions**

REFERENCE permission on the source object and ALTER permission on the target object.

**Examples**

The following example shows how to create a PAGE compressed index on the table ‘dbo.Staging\_Fact’ that is otherwise identical to the index ‘IX\_Fact\_2’ on the table ‘dbo.Fact’. The index will be created in the file group currently mapped to partition 10 of the partition scheme.

|  |
| --- |
| DECLARE @objid int, @indid int;  SELECT @objid = object\_id, @indid = index\_id FROM sys.indexes WHERE object\_id = OBJECT\_ID(’dbo.Fact’) AND name = ’IX\_Fact\_2’;  EXEC ptf.CreateIndex @source\_tab\_id = @objid,  @index\_id = @indid,  @target\_object = ’dbo.Staging\_Fact’,  @part\_num = 10,  @compression = 2; |

**See also**

For each indexed view that references the source table an equivalent indexed view is created that references the target table.

**Syntax**

|  |
| --- |
| CreateIndexedViews [ @source\_tab\_id = ] *source\_table\_id* , [ @target\_table = ] ’*table\_name*’ { , [ @part\_num = ] *partition\_number* ] [ , [ @compression = ] *compression\_mode* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_tab\_id** = ] *source\_table\_id*

Is the object ID of a table in the current database. *source\_table\_id* is **int**, with no default. All indexed views that reference this table are identically being created for the target table.

[ **@target\_table** = ] '*table\_name*'

Is the name of the table the new indexed views will reference instead. The table must already exist. *table\_name* is **nvarchar(256)**, with no default. The target table and the source table must belong to the same database schema.

[ **@part\_num** = ] *partition\_number*

If the source table is partitioned, this is the number of the partition which determines the file group the indexed views will be created in. *partition\_number* is **int**, with a default of NULL. *partition\_number* must be a valid partition number within the partition scheme, or 1 or NULL for a non-partitioned table.

[ **@compression** = ] *compression\_mode*

Defines the compression to be applied to the indexed views. *compression\_mode* is **int**, with a default of NULL. Valid values are 0, 1, 2, 3, 4 or NULL. Specifying NULL causes the index being compressed the same way as the corresponding partition of the source index.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

For all indexed views that reference the source table an equivalent indexed view will be created that references the target table instead of the source table in its view definition along with the same indexes that exist on the source view.

If *partition\_number* is specified, the source table must be partitioned and the file group each indexed view will be created on, is determined by the partition scheme mapping for the given *partition\_number*.

This procedure must parse the definition of the views in order to replace all references to the source table in the view definition with references to the target table. Since this parsing is only performed in a very simplistic way, it is not guaranteed to work for all possible view definitions. Thus, it is advisable to run this procedure first with @debug set to 2 to verify the generated output for valid CREATE VIEW statements.

**Permissions**

REFERENCE permission on the source table and all the indexed views that reference this table, as well as REFERENCE permissions on all other objects referenced by the views, plus ALTER permission on the target database schema.

**Examples**

The following example shows how to duplicate all indexed views that reference table ‘dbo.Fact’ with views that reference the table ‘Staging\_Fact’ instead. All the indexes will be materialized in the file group that is determined by the scheme the source table is partitioned with and the specified partition number.

|  |
| --- |
| DECLARE @objid int; SET @objid = OBJECT\_ID(’dbo.Fact’);  EXEC ptf.CreateIndexedViews @source\_tab\_id = @objid,  @target\_table = ’Staging\_Fact’,  @part\_num = 10; |

**See also**

Duplicates the source table’s indexes, foreign key and check constraints, and indexed views on the target table.

**Syntax**

|  |
| --- |
| CreateIndexesAndConstaints [ @source\_tab\_id = ] *source\_table\_id* [ , [ @target\_table = ] ’*table\_name*’ ] [ , [ @fg\_name = ] ’*filegroup\_name*’ ] [ , [ @part\_num = ] *partition\_number* [ , [ @part\_scheme = ] ’*partition\_scheme\_name’* ] [ , [ @part\_col\_id = ] *partition\_column\_id* ] [ , [ @exclude\_check = ] ’*constraint\_name*’ ] [ , [ @compression = ] *compression\_mode* ] [ , [ @compress\_all = ] *all\_flag* ] [ , [ @create\_indexes = ] *index\_code* ] [ , [ @incremental\_stats = ] *stats\_flag* ] [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_tab\_id** = ] *source\_table\_id*

Is the object ID of a table in the current database. *source\_table\_id* is **int**, with no default.

[ **@target\_table** = ] '*table\_name*'

Is the name of the table on which the indexes and constraints shall be created. *table\_name* is **nvarchar(256)**, with a default of NULL. If no target table has been specified, the target is the source table, in which case the table and all its indexes are going to be partitioned.

[ **@part\_num** = ] *partition\_number*

If the source table is partitioned, this is the number of the partition where the target table later will be switched in. *partition\_number* is **int**, with a default of NULL. If the *filegroup* *\_name* is specified, this parameter is being ignored. *partition\_number* must be a partition of the source table and all the partition schemes the indexes of the source table are based on must have a file group mapped to it that is not read only.

[ **@part\_num** = ] *partition\_number*

If the source table is partitioned or if **@ps\_id** is specified, this is the number of the partition which determines the file group the indexes will be created in. *partition\_number* is **int**, with a default of NULL. *partition\_number* must be a valid partition number within the partition scheme, or 1 or NULL for a non-partitioned table.

[ **@ps\_id** = ] *partition\_scheme\_id*

Is the ID of a partition scheme in the current database. *partition\_scheme\_id* is **int**, with a default of NULL. If the source table is not partitioned, but the table’s file group shall be determined by a partition mapping, this parameter specifies the partition scheme to be used and **@part\_num** the partition used for the mapping.

[ **@part\_scheme** = ] ‘*partition\_scheme\_name’*

Is the name of an existing partition scheme. This scheme will be used to partition the target table and its indexes. *partition\_scheme\_name* is **sysname**, with a default of NULL.

[ **@part\_col\_id** = ] *partition\_column\_id*

If a *partition\_scheme\_name* has been specified, this is the ID of the column that will be used to partition the indexes. The data type of this column must be compatible with the type of the partition function the specified partition scheme is based on. *partition\_column\_id* is **int**, with a default of NULL.

[ **@exclude\_check** = ] '*constraint\_name*'

Is the name of an existing column level check constraint on the source table. If *constraint\_name* is specified, all check constraints defined on the source table, except this one, will be duplicated on the target table. *constraint\_name* is **sysname**, with a default of NULL.

[ **@compression** = ] *compression\_mode*

Defines the compression to be applied to the indexes and indexed views. Valid values are 0, 1, 2, 3, 4 or NULL. NULL causes the indexes being compressed the same way as the corresponding partition of the source index. *compression\_mode* is **int**, with a default of NULL.

[ **@compress\_all** = ] *all\_flag*

Is 0 or 1, with a default of 0. If 0 is specified, the compression mode will only be applied to the table or its clustered index, otherwise all indexes and indexed views will be compressed accordingly.

[ **@create\_indexes** = ] *index\_code*

Is 1, 2 or 3, with a default of 3. If 1 is specified, only a clustered index will be created, with 2 also all nonclustered indexes will be created, and with 3 also a nonclustered columnstore index if one exists on the source table.

[ **@incremental\_stats** = ] *stats\_flag*

Is 0 or 1, with a default of 0. If 1 is specified, and the target table is partitioned the corresponding indexes on the target table will be created with incremental statistics turned on. On SQL Server 2012 and below this parameter is being ignored.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure involves several steps. First, all clustered, nonclustered, or columnstore indexes that exist on the source table are being created identically on the target table. The only index that may already exist on the target table is a clustered index. If *compression\_mode* is not NULL and if *all\_flag* is set to 1, the compression mode is applied while creating all the indexes, otherwise, the compression mode is only applied while creating the clustered index, or alternatively, if the clustered index already exists or the source table is not clustered, only the table is being compressed.

Next, for all indexed views that reference the source table a corresponding indexed view is being created that references the target table along with all accompanying indexes.

Then, all foreign key constraints and check constraints that exist on the source table are being created identically on the target table. The only check constraint that is being exempted, is the one specified in **@exclude\_check** parameter.

**Permissions**

REFERENCE permission on the source table and ALTER permission on the target table.

**Examples**

The following example shows how to create on the table ‘Staging\_Fact’ all indexes and constraints, except the check constraint ‘check\_date’, identically to the indexes and constraints on the source table ‘dbo.Fact’. The file group the indexes will be created in is defined by the mapping of partition 10 of the source table’s partition scheme. All indexes will be ROW-compressed.

|  |
| --- |
| DECLARE @objid int SET @objid = OBJECT\_ID(’dbo.Fact’);  EXEC ptf.CreateIndexesAndConstraints @source\_tab\_id = @objid,  @target\_table = ’Staging\_Fact’,  @part\_num = 10,  @exclude\_check = ’check\_date’,  @compression = 1,  @compress\_all = 1; |

**See also**CreateIndex  
CreateIndexedViews

Performing all partition related actions on a partitioned table.

**Syntax**

|  |
| --- |
| SwitchPartition2 [ @source\_tab\_id = ] *partitioned\_table\_id*  [ , [ @archive\_table = ] ’*archive\_table\_name*’ ] [ , [ @out\_part = ] *output\_partition\_number* ] [ , [ @staging\_table = ] ’*staging\_table\_name*’ ] [ , [ @in\_part = ] *input\_Partition\_number* ][ , [ @split\_value = ] *boundary\_value* ] [ , [ @new\_filegroup = ] ’*filegroup\_name*’ ] [ , [ @check\_name = ] ’*constraint\_name*’ ] [ , [ @upper\_bound = ] *boundary\_value* ][ , [ @make\_readonly = ] *readonly\_flag* ] [ , [ @keep\_indexes = ] *out\_index\_type* ] [ , [ @no\_merge = ] *no\_merge\_flag* ] [ , [ @no\_drop = ] *no\_drop\_flag* ] [ , [ @wait\_option = ] *’option\_string’* [ , [ @temp\_mask = ] *temporal\_mask* [ , [ @archive\_mode = ] *archive\_mode* [ , [ @debug = ] *debug\_flag* ] |

**Arguments**

[ **@source\_tab\_id** = ] *partitioned\_table\_id*

Is the object ID of a partitioned table where the actions will take place. *partitioned\_table\_id* is **int**, with no default.

[ **@archive\_table** = ] '*archive\_table\_name*'

Is the name of the archive table where the output partition will be switched to. This table must not yet exist, as it will be created by this procedure as an exact copy of the partitioned table in the file group that is mapped to the output partition. If this file group is read only, it will be changed to read/write prior to this step. Depending on the *out\_index\_type* setting this table will be just clustered or will have all indexes of the partitioned table. The archive table will be compressed the same way as the output partition. *archive\_table\_name* is **nvarchar(256)**, with a default of NULL. If *archive\_table\_name* is NULL, no partition will be switched out.

[ **@out\_part** = ] *output\_partition\_number*

Is the number of the partition that will be switched out into the archive table. *output\_partition\_number* is **int**, with a default of NULL. If *archive\_table\_name* is specified, *output\_partition\_number* must be a valid partition number of the partitioned table. After the partition has been switched out, this partition will be removed by a MERGE operation, except if *no\_merge\_flag* is set to 1.

[ **@staging\_table** = ] '*staging\_table\_name*'

Is the name of the non-partitioned staging table to be switched in. Before being switched in, this table needs to be prepared for switching. The target partition will adopt the compression setting of the staging table. *staging\_table\_name* is **nvarchar(256)**, with a default of NULL. If *staging\_table\_name* is NULL, no table will be switched in. This table will be dropped after it has been successfully switched in, except if *no\_drop\_flag* is set to 1.

[ **@in\_part** = ] *input\_partition\_number*

Is the number of the partition where the staging table will be switched in, or the number of the partition where the filegroups mapped to it will be made READ\_ONLY . *input\_partition\_number* is **int**, with a default of NULL. If *staging\_table\_name* is specified or *readonly\_flag* is 1, *input\_partition\_number* must be a valid partition number of the partitioned table. If this partition is the target of a switch operation, it must not contain any data (except when operating in swap mode, i.e. the input partition is identical to the output partition).

[ **@split\_value** = ] *boundary\_value*

If *boundary\_value* is specified, a new partition will be created. The specified value is the new boundary value added to the partition function of the partitioned table by a SPLIT operation. To be able to split a partition, the partition must be empty on all objects that are partitioned by partition schemes that are dependent on the same partition function. If *staging\_table\_name* is also specified, the split takes place before the staging table is being switched in. *boundary\_value* is **sql\_variant** with a base type compatible with the type of the partitioning column, with a default of NULL. This parameter cannot be used in swap mode.

[ **@new\_filegroup** = ] '*filegroup\_name*'

Is the file group to be mapped to the new partition created by **@split\_value**. This file group will only be mapped to the primary partition scheme, i.e. the partition scheme the base table is partitioned with. If also secondary partition schemes exist, the mappings for those partition schemes must be set explicitly in advance by means of “ALTER PARTITION SCHEME NEXT USED” statements. If no *filegroup\_name* is specified, the file groups the new partition is mapped to are being determined either by the file groups that have been mapped to the output partition, or, if no partition is being switched out, by the file groups mapped to the partition preceding the newly created partition. Any file group that will be mapped to the new partition and that is read only will be changed to read/write. *filegroup\_name* is **sysname**, with a default of NULL.

[ **@check\_name** = ] '*constraint\_name*'

If *constraint\_name* is specified, a column level check constraint with that name will be created on the partitioned table for the partitioning column. If a check constraint with that name already exists, that constraint will be dropped before the new one is being added. *constraint\_name* is **sysname**, with a default of NULL.

[ **@upper\_bound**  = ] *boundary\_value*

Is the upper bound used for the check constraint. *boundary\_value* is **sql\_variant** with a base type compatible with the type of the partitioning column, with a default of NULL. If *constraint\_name* is specified, also *boundary\_value* must be specified. The actual check condition used depends on the direction of the partition function. For RANGE LEFT the condition is “p <= *boundary\_value*”, and for RANGE RIGHT it is “p < *boundary\_value*”, where p is the partitioning column.

[ **@make\_readonly** = ] *readonly\_flag*

Is 0 or 1, with a default of 0. If 1 is specified, all file groups mapped to the partition specified in **@in\_part** and that are read/write will be changed to read only.

[ **@keep\_indexes** = ] out\_*index\_type*

Is 0, 1, 2 or 3, with a default of 1. This value defines which indexes will be preserved on the archive table. If 0 is specified, the archive table will not have any indexes, if it is 1, it will only have a clustered index if the partitioned table also is clustered, with 2 and 3 the archive table will be fully prepared with all nonclustered indexes and with 3 also a with a nonclustered columnstore index. *index\_code* is **tinyint**.

[ **@no\_merge** = ] *no\_merge\_flag*

Is 0 or 1, with a default of 0. If 1 is specified and *archive\_table* is not NULL, the partition that has been switched out will not be removed. Since the first partition of a RANGE RIGHT partition or the last partition of a RANGE LEFT partition cannot be merged, *no\_merge\_flag* must be 1 if one of those partitions is being switched out. In swap mode, this parameter must be 1. No merge can be performed if any other objects exist that are partitioned by the same partition schemes and where the output partition is not empty.

[ **@no\_drop** = ] *no\_drop\_flag*

Is 0 or 1, with a default of 0. If 1 is specified and *staging\_table* is not NULL, the staging table will not be dropped after it has been switched in.

[ **@wait\_option** = ] *‘option\_string’*

Starting with SQL Server 2014, one can add an option to a table switch statement that defines the actions to take when the statement is blocked by a query on the partitioned table running in parallel. The string specified must have the following syntax (see BOL for details):  
WAIT\_AT\_LOW\_PRIORITY (MAX\_DURATION = <number of minutes to wait>,  
ABORT\_AFTER\_WAIT = {NONE | SELF | BLOCKERS})  
The parameter is of type **nvarchar(80)**, with a default of NULL.

[ **@temp\_mask** = ] *temporal\_mask*

Is 0, 1, 2 or 3, with a default of 2. If *archive\_mode* is 1 and if the source table is a temporal table and applying the mask to the type of temporal table (as stored in temporal\_type column of sys.tables catalog view) returns a nonzero value, additional statements will be added that are necessary to switch out data from a temporal.

[ **@archive\_mode** = ] *archive\_mode*

Is NULL, 0, 1, or 2 with a default of NULL. NULL defaults to the value 0, if *archive\_table\_name* is NULL, and 1 otherwise. A value of 0 means do nothing with the next output partition, a value of 1 causes the next output partition to be archived in the table specified in *archive\_table\_name*, and a value of 2 specifies truncation of the data in the next output partition without archiving.

[ **@debug** = ] *debug\_flag*

Is 0, 1 or 2, with a default of 0. If 2 is specified, only the T-SQL statements that would be executed are being printed. If 1 is specified all statements will be executed, but additionally to logging in PTF’s Log table, the log entry also will be printed.

**Return Code Values**

0 (success) or 1 (failure).

**Result Sets**

None.

**Remarks**

This procedure either operates in normal mode, or in swap mode. In swap mode, the input and output partitions are identical, i.e. before the staging table is being switched into the target partition, this partition is being switched out into the archive table. Not all operations are available in swap mode, e.g. no new partition can be created, and the target partition cannot be removed. Hence, the **@split\_value** and **@new\_filegroup** parameters should not be used in this mode and **@no\_merge** must be set to 1.

All operations performed by this procedure are meta-data only operations and thus should complete within seconds, unless the operations are being blocked by existing locks on the partitioned table.

**Permissions**

Requires ALTER and ALTER ANY DATASPACE permission on the database.

**Examples**

The following example shows how to switch out partition 2 of the partitioned table ‘dbo.Fact’ into table ‘Archive\_Fact’ and then dropping partition 2, switch in the already prepared table ‘Staging\_Fact’ into partition 14 and dropping this table afterwards, create a new partition with the new boundary value ‘2016-07-01’ and map this new partition to filegroup ‘FG\_July’, recreate the check constraint ‘check\_date’ with the check condition (create\_date < ’2016-07-01’) where create\_date is the name of the partitioning column, and make the file group where the table ’Staging\_Fact’ did reside in READ\_ONLY. All this is being accomplished in a single stored procedure call:

|  |
| --- |
| DECLARE @objid int; SET @tabid = OBJECT\_ID(’dbo.Fact’, ‘U’);  EXEC ptf.SwitchPartition2 @part\_table\_id = @tabid,   @archive\_table = ’Archive\_Fact’,  @archive\_mode = 1,  @out\_part = 2,   @staging\_table = ’Staging\_Fact’,  @in\_part = 14,   @split\_value = ’2016-07-01’,  @new\_filegroup = ’FG\_July’,  @check\_name = ’check\_date’,  @upper\_bound = ’2016-07-01’,  @make\_readonly = 1; |

**See Also**