# Intro

If you are working with data warehousing or reporting, you’ll recognize this problem as a recurring headache whenever you’re designing an ETL process for fact tables:

If you want to completely reload all the rows of a fact table, you will typically start by emptying (or truncating) the fact table, and then load new data into it. But during the loading process, depending on what your job does, there will not be any data in the table, or worse, it will be half-filled and incorrect.

Worst-case:

If your ETL job crashes, the table will remain empty. Now, if your ETL job takes an hour to run, that is a problem.

There are a few different approaches to solving this problem. I will focus on the Partition Switching approach. Alternatives will be mentioned in the last section.

## Partition Switching in General

If you have worked with partitioned tables, partition switching may have come to mind. When moving a partition between tables, SQL Server validates an extensive set of rules to ensure that the replacement data set is compatible with its predecessor; for example column names, order, data types, nullability, primary keys, foreign keys, clustered indexes must be identical or the switch fails. Also, since a partition switch is not swapping the table itself, but rather a partition under the table, key table metadata, like object ID, are unaffected.

Partition switching can be performed on non-partitioned and non-partitioned tables and **does not necessarily require Enterprise or Developer edition**.

Why? Because behind the scenes, SQL Server tables *always* store their data in partitions. Enterprise and Developer editions add the option of splitting a table’s data into multiple partitions; however, all SQL Server editions auto-create a single partition per ordinary, unpartitioned table to hold its data.

## Partition Switching on partitioned tables

**(Does require Enterprise edition)**

**Partitioning** is a very powerful feature of SQL Server that allows you to store table rows in multiple “partitions”. These partitions look like a bunch of separate tables with identical column and index definitions, except the partitioning is completely transparent, so you only see one table. One of the greatest advantages of partitioning is the ability to distribute different partitions on different physical filegroups, which allows you to split the I/O workload of the table on different physical storage units. But arguably, one of the most useful and beautiful features of partitioning is **partition switching**.

Think of partitions as slices of bread if you will. Each slice can be different in content (rye, wheat) and can have different thickness, but the column definitions are obviously always the same. When you query the table, SQL Server will try to find specific partitions that contain the information you’re looking for in a process called **partition elimination**. Probably the most common application of partitioning is for date-derived expressions; a partition can for instance contain a single month, quarter, or year of data.

**Partition switching** is when you swap (switch) the data in two table partitions with each other. Because this process does not actually move any data (the partitions/tables must be located on the same physical filegroup), this is a lightning-fast process.

Here is s how it works:

***ALTER TABLE dbo.aPartitionedTable***

***SWITCH PARTITION 3 TO dbo.aNonPartitionedTable\_prev;***

The example query above takes all records from partition 3 in a partitioned table and switches them into the non-partitioned table, which has to be empty in order for this to work. To switch data back into the partition, you can use the following:

***ALTER TABLE dbo.aNonPartitionedTable\_Stage***

***SWITCH TO dbo.aPartitionedTable PARTITION 3;***

You can even move data between two partitioned tables:

***ALTER TABLE dbo.aPartitionedTable***

***SWITCH PARTITION 3 TO dbo.anotherPartitionedTable PARTITION 12;***

In the case of loading a fact table, you can use a staging table to load your data into and then switch that staging table into the actual fact table (one partition at a time).

There are a bunch of other [requirements](http://technet.microsoft.com/en-us/library/ms191160.aspx) associated with partition switching, but they’re outside the scope her.

Here is an example on how you could implement partition switching to load a single partition of fact data while keeping the fact table online and populated the entire time:

--- Populate staging table - this is the part of the ETL job that

--- will take up pretty much all of the time.

***TRUNCATE TABLE STAGE.sales;***

***INSERT INTO STAGE.sales (...)***

***SELECT (...) FROM (...);***

--- Clearing the partition by switching it to a dummy table. This is

--- required because the target of a partition switch must be empty.

--- You could DELETE these rows, but this is much, much, faster.

***ALTER TABLE FACT.sales PARTITION 3***

***SWITCH TO STAGE.sales\_empty;***

--- .. then clear the dummy table:

***TRUNCATE TABLE STAGE.sales\_empty;***

--- Finally, switch the new staged data into the fact table:

***ALTER TABLE STAGE.sales***

***SWITCH TO FACT.sales PARTITION 3;***

This method has another obvious advantage over the traditional method of just emptying the fact data and re-populating it:

If the ETL job fails, you will still have the old fact data intact and untouched, giving you a bit of peace to try to sort out what went wrong with the ETL task, without the screaming users and all that comes with those.

## Partition Switching on non-partitioned tables

**(Does not require Enterprise edition)**

Given, partitioning is not for everyone. It can add quite a deal of complexity to your solution. Some of the issues are obvious and easily managed, others are not.

But SWITCH command can still be used on *all editions* of SQL Server, as mentioned before.

You won’t be able to partition a table and then reload/switch those individual partitions, but you can still switch entire tables.

Below is a partition switching example, but without the partitioning bit:

-- Populate staging table - this is the part of the ETL job that

--- will take up pretty much all of the time.

***TRUNCATE TABLE STAGE.sales;***

***INSERT INTO STAGE.sales (...)***

***SELECT (...) FROM (...);***

--- Clearing the fact table by switching to another table. This is required because the target

--- of a partition switch must be empty.

***ALTER TABLE FACT.sales***

***SWITCH TO FACT.sales\_prev;***

--- Switch the new staged data into the fact table:

***ALTER TABLE STAGE.sales***

***SWITCH TO FACT.sales;***

As with partition switching, the source and target of the switch operation still need to have identical column and index definitions. And apart from having a partitioned fact table, you will still enjoy the other advantages, such as still having a populated fact table if the ETL job crashes half-way.

The downside is that you need to populate the entire fact table, which requires time and disk space.

## When switching won’t work

Switching tables or partitions won’t work in a number of scenarios, including the following:

* when you don’t have enough space to build the stage table on the same file group, because switching always has to happen within the same filegroup.
* when one of the switched tables is replicated.

There are more requirements: [Transferring Data Efficiently by Using Partition Switching](http://technet.microsoft.com/en-us/library/ms191160.aspx)

## Other approaches

There exists 2 other methods to accomplish the type of switching, but considering the flexibility of switching and the fact that it’s available on Standard Edition as long as you’re not working with partitions, I would still recommend Partition Switching.

* Renaming Tables

Pro:

* Inheritance of index and statistics
* Effective and fast table swap
* No need for identical column and index definitions

Cons:

* No metadata changes (Object ID contains the same)
* Requires necessarily additional step of updating the system’s cache
* Schema Swap Transfer

Pro:

* Effective and fast table swap
* Metadata changes (Object ID’s are different)
* No need for identical column and index definitions

Cons:

* Column store indexes will have to be disabled before DML operations and then enabled at the end
* NO Inheritance of index and statistics
* ALTER SCHEMA TRANSFER locks the whole schema (SCH-M lock)
* Potential High Risk of Schema lock’s on big tables, requires SQL Server service restart
* Swap table needs to be recreated