# Partition indexes for improved SQL Server 2005 performance

| Source | http://searchsecurity.techtarget.com/tip/0,289483,sid87_gci1206729,00.html |
|---|---|

## Partition indexes for improved SQL Server 2005 performance

**Index partitioning is one of a number of new features introduced in SQL Server 2005 as a way to distribute the load for a given index across multiple files, which can enhance parallelism or improve index performance in other ways.**

**TABLE OF CONTENTS**



### SQL Server 2000 partitioned views vs. SQL Server 2005 index partitioning

**Earlier editions of SQL Server accomplished index partitioning with partitioned views. Queries to and changes against tables could be constrained in certain ways by a view, so only needed physical files would be queried or modified. For instance (this is an arbitrary example but it suits our needs), if you had 26 tables for a customer database, one for each letter of the alphabet, you could use a partitioned view to aggregate the results from all of the tables and use WITH CHECK constraints to update only the needed tables. You could run a query against all customers whose names begin with "B" and the partitioned view would know only to poll the "B" table.**

**The downside of partitioned views is that they must be created and managed manually. In SQL Server 2005, there's greater abstraction between partitions, tables and data, so they can be manipulated independently.**

**Also new is index partitioning, where the index (or indexes) for a given table are partitioned or constrained across multiple files and filegroups. Here I've assembled some basic guidelines for how to set up and use index partitioning; the exact details for how to do this are described in SQL Server 2005 Books Online.**

### Create an index with partitioned data

**There are two ways to create an index with partitioned data: Partition the index as the data is partitioned or partition the index separately. Which partition scheme to use should depend on how your data is accessed and updated.**

**In the first setup your index is *"partition aligned."* By default any newly-created index on a partitioned table will have the same partitioning as the table itself. This is best if you:**

- **know that a great deal of data will be inserted into the table over time;**
- **anticipate adding partitions as you go;**
- **and believe that these are the most important aspects of your data setup for this table.**

**Take the example where a partition arrangement covers a year of data split up according to months, where the date is used as a primary key. Partitioning the index this way speeds up date lookups since SQL Server can quickly determine where a given key may be in the index.**

**There are times you won't always want to use a partition-aligned arrangement, possibly in cases where you have a unique index key that does not contain the table's partitioning column. For instance, if we were using the above A to Z partitioning scheme but the index key for the tables was a GUID or auto-increment number rather than the customer name, you could keep the index in its own partition so it's not aligned with the table. In any case if you explicitly put the index on a different filegroup, partitioning will not match the table.**

**If you're partitioning data that has a unique index, the column used for partitioning be the same used for the unique index key. If your unique partition index is a client ID number, for instance, that will be the same column used to partition the index key as well.**

### Understand partition function and partition scheme

**Partitions are made up of two things: *partition function* and *partition scheme*. The first represents how data itself is split across different partitions. For instance, in the A to Z example, the data is partitioned according to each letter of the alphabet as 26 separate partition functions.**

**A *scheme* represents how each partition in the partition function is mapped to a filegroup. If our A to Z table has "A" data stored in a physical file in one filegroup and the "A" index stored in another physical file in the same filegroup, a partition-aligned index can help speed up and parallelize access to both the data and indexes. That way multiple CPUs can**

**work on different partitions or physical files. (You can parallelize things further by placing indexes and data on separate physical spindles, if you have them.)**

### Account for tempdb space

**Building partition-aligned indexes takes memory and uses space in tempdb. Many database administrators don't set tempdb's space allotments to anything beyond the default when they install SQL Server, and the time and effort it takes to auto-expand tempdb can put a crimp in performance. Also, indexes that are partitioned differently are built using different memory allocation schemes: a partition-aligned index is built with one sort table at a time, while a nonaligned index is built with all its sort tables at the same time.**

**Microsoft states in Books Online that the minimum size for each sort table per partition is 40 pages of 8 KB per page, so a nonaligned partitioned index with 26 partitions (using the previous A to Z example) would require 1,040 pages -- approximately 4.25 MB of memory. A nonaligned index would only need 163,840 bytes. For the most robust SQL Server installations this shouldn't be a problem, but be mindful if you're dealing with extremely large partitioning schemes *and* working with multiple partitioning schemes at one time.**

***~~~ End of Article ~~~***