|  |  |
| --- | --- |
| |  | | --- | | **https://gallery.mailchimp.com/9082566fb63d87be35c0662bc/images/a043049c-3eda-407d-9b3b-088d74c50b04.png** | |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | |  | | --- | | Are Partitioned Tables Right for You? In the last episode, I talked about [knowing when it’s time to scale out:](https://www.brentozar.com/archive/2019/08/dba-training-plan-16-is-it-time-to-scale-out/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) identifying when our data is getting to be so large that we have to split it across multiple servers, and I explained why that is so challenging. But what about table partitioning – SQL Server’s ability to break up a single table into smaller ones on the same server?  Great volumes have been [written about table partitioning](https://www.brentozar.com/table-partitioning-resources/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d). It’s a complex feature and you can read for days to just understand how you might apply it. But will it improve performance for you? Table partitioning produces great benefits for some applications, but causes giant headaches for others. What Table Partitioning Does Table partitioning allows tables or indexes to be stored in multiple physical sections: a partitioned index is like one large index made up of multiple little indexes. Each chunk, or partition, has the same columns – just a different range of rows. In theory, SQL Server handles this transparently for you: a partitioned heap, clustered index, or non-clustered index can be referenced as a single structure although it’s stored in independent physical partitions. In other words, in theory you don’t need to change any code in the calling applications. (Is this true in reality? More on this later.)  Let’s say we want to partition the Stack Overflow Posts table – that’s where the questions and answers are stored. Every question & answer is considered a Post. I’m going to use the 50GB 2013 version of [the Stack Overflow database](https://www.brentozar.com/archive/2015/10/how-to-download-the-stack-overflow-database-via-bittorrent/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) because the Posts table is starting to grow: it’s 17,142,169 rows taking up 37GB space on disk – that’s most of the database, so breaking it up will be a performance boost, right?  Let’s say we want to partition them by CreationDate, and we want to store each year’s data in a separate file. We’ll need to start by creating filegroups for each year:  USE [master]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2008]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2008A',  FILENAME = N'M:\MSSQL\Data\History\_2008A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2008]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2009]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2009A',  FILENAME = N'M:\MSSQL\Data\History\_2009A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2009]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2010]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2010A',  FILENAME = N'M:\MSSQL\Data\History\_2010A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2010]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2011]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2011A',  FILENAME = N'M:\MSSQL\Data\History\_2011A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2011]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2012]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2012A',  FILENAME = N'M:\MSSQL\Data\History\_2012A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2012]  GO  ALTER DATABASE [StackOverflow2013] ADD FILEGROUP [History\_2013]  GO  ALTER DATABASE [StackOverflow2013] ADD FILE ( NAME = N'History\_2013A',  FILENAME = N'M:\MSSQL\Data\History\_2013A.ndf' , SIZE = 524288KB , FILEGROWTH = 524288KB )  TO FILEGROUP [History\_2013]  GO  You’ll notice in this example that I’m using the same drive (M) for all of the files – in partitioning, though, it’s common to put the different years of data in different drives, and even stripe the data across multiple files. Let’s keep it simple for this demo though.  Next, I need to create a partition function and a partition scheme to tell SQL Server which data belongs on which drive/file:  USE StackOverflow2013;  GO  CREATE PARTITION FUNCTION pf\_CreationDate ( DATETIME )  AS RANGE RIGHT FOR VALUES ( '20090101',  '20100101',  '20110101',  '20120101',  '20130101' );  CREATE PARTITION SCHEME ps\_CreationDate  AS PARTITION pf\_CreationDate TO  (History\_2008, History\_2009, History\_2010,  History\_2011, History\_2012, History\_2013 );  Partitioning pros will note that I’m violating all kinds of best practices. Partitioning pros should also note that I’m trying to teach the basics of partitioning in a really short amount of space. Partitioning pros should go write their own blog posts.  Finally, we move the table over onto the new storage structure:  /\* Have to drop the existing clustered index before we can add a new one: \*/  ALTER TABLE dbo.Posts DROP CONSTRAINT PK\_Posts\_Id;  /\* Create a new partitioned one - note for now it's not the primary key: \*/  CREATE UNIQUE CLUSTERED INDEX CLIX\_CreationDate\_Id  ON dbo.Posts(CreationDate, Id) ON ps\_CreationDate(CreationDate); To pull that off, I had to drop the primary key. This table – like most tables – has a clustered primary key, and SQL Server doesn’t let me just move the primary key over to the new storage structure. Instead, I have to:   * Remove any foreign key relationships that reference this table * Drop the primary key * Create a new clustered index on the partition scheme (which splits the data across multiple files) * Create a primary key again * Recreate the foreign key relationships   For more details about why I had to do all that, see [this excellent Stack answer by Remus Rusanu](https://stackoverflow.com/a/7655495/26837?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d), and make sure to read the linked [Microsoft article, Special Guidelines for Partitioned Indexes](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms187526(v=sql.105)?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d).  See, partitioned indexes come with a whole lot of gotchas:   * What I just did would have involved a lot of transaction log activity and ugly downtime * A real-life implementation requires [reading the documentation (and it’s big)](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2005/administrator/ms345146(v=sql.90)?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) * You really need to [load data with partition switching](https://www.red-gate.com/simple-talk/sql/database-administration/gail-shaws-sql-server-howlers/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) * To do that, your nonclustered indexes need to be aligned (use the partition scheme) * [Fixing non-aligned indexes requires dropping & recreating them](https://www.brentozar.com/archive/2018/10/fixing-nonaligned-indexes-on-partitioned-tables/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) * When handling concurrency, [locking on partitioned tables is tricky](https://www.brentozar.com/archive/2017/11/partition-level-locks-confusing/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) * [Merging & switching partitions can have surprise effects](https://www.brentozar.com/archive/2013/01/best-practices-table-partitioning-merging-boundary-points/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d) * If the WHERE clause of your query doesn’t include the partitioning key, [query performance usually gets worse](https://www.brentozar.com/archive/2015/09/why-is-this-partitioned-query-slower/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d)   <sigh>  Partitioned tables work, but…they’re a lot of hard work. So should you partition your tables? If you’re dealing with a 1TB+ data warehouse that can do sliding window loads, possibly.  If you’re dealing with a smaller one, or you can’t do sliding window loads, [partitioned views are usually a better fit](https://www.brentozar.com/archive/2016/09/partitioned-views-guide/?mc_cid=40c20335ef&mc_eid=%5bUNIQID%5d).  If you’re doing transactional workloads, and you can’t guarantee that the partitioning key will be in most of your WHERE clauses, then no, partitioning isn’t a good fit. Plain ol’ index and query tuning will be a better match. | | | |