Optimize tempdb in SQL Server by striping and splitting to multiple files

Serdar Yegulalp

The practice of storing a database across multiple files in a filegroup (or across multiple filegroups) is done all the time to enhance the performance of user-created databases. What, then, about doing the same thing to increase the performance of the temporary database, **tempdb**?

On SQL Server systems with a single logical processor or relatively small load, a single file may really be all you need to handle **tempdb**. But on larger, more heavily-trafficked systems, **tempdb** performance can benefit from having multiple physical files allocated to it. Here are some pointers on how to go about doing this most effectively.

1. **Create one physical file per processor**. The processors in question can be either physical or virtual, but the point is to provide one separate physical file per worker whenever possible. This way, in round-robin fashion, each worker has a separate physical file, which reduces contention of resource allocation. See blog on article on tempdb by the PSS SQL Server Engineers at Microsoft for some information about this behavior in context.
2. **Pre-allocate a decent amount of space per file**. This is done so there will always be a decent amount of space already allocated for the **tempdb** files. You should have auto-grow turned on for **tempdb**'s files, but there's a balancing acting between how much to allow the **tempdb** files to grow and how much time the system should spend allocating for growth.

   Microsoft's own documentation about optimizing **tempdb** performance talks about this and indicates that the pre-allocated size you choose for tempdb should cover the vast majority of work scenarios and that autogrowth should only kick in under the most extreme circumstances. The SQL Server Scalability FAQ suggests that if you're dealing with terabyte-size databases, expect to allocate more than 100 GB for each tempdb file. So, the exact size of the files is going to vary enormously depending on what grade of work you're doing.

   The single best way to figure out how much space to set aside is to use statistics harvested from a live workload to gauge **tempdb** usage. You can pre-allocate a great deal at first, then monitor **tempdb** usage to get an idea of exactly how much of the space is being used.

3. **Make sure the tempdb files are contiguous.** This improves linear read performance, of course, but it is generally not something to obsess about. The best way to insure that the files are contiguous is to create them, then use a utility like **CONTIG** to before putting the system into service. Since **tempdb** files may take a long time to make contiguous, if **CONTIG** reports that the file is only in a few fragments (i.e., there are no individual pieces smaller than 64 MG), the performance impact will most likely be minimal. In this case, the file probably isn't worth defragmenting.

4. **Place each tempdb file on a separate logical unit / physical disk spindle whenever possible.** The same advice applies to physical files on user databases as well. This parallelizes the way **tempdb** is accessed, so there's less contention on the hardware level for any particular file.
5. **The tempdb files should not share space with the user databases.** Again, this is to reduce contention and enhance parallelism; **tempdb**'s files should always be on their own spindles whenever possible. Even if those multiple files are on the same physical disc (if you have no other choice), they can still increase performance. The more pre-allocated space there is available, the less contention there will be to allocate that space to begin with.
6. **Consider using trace flag 1118 to reduce tempdb contention, but don't expect a magic bullet.** Turning on trace flag 1118 in SQL Server 2005 is alleged to improve performance in situations where there are multiple **tempdb** files. But it is definitely one of those optimizations where the end user's mileage will vary. Linchi Shea of SQLBlog.com discussed the issue and then ran tests with flag 1118 on and off in a variety of scenarios. He concluded that the best approach is to test using your live workload in order to determine whether or not it needs to be enabled. (The differences, to me, don't appear to be terribly dramatic to begin with.)