[**http://sqlcat.com/technicalnotes/archive/2009/09/22/resolving-pagelatch-contention-on-highly-concurrent-insert-workloads-part-1.aspx**](http://sqlcat.com/technicalnotes/archive/2009/09/22/resolving-pagelatch-contention-on-highly-concurrent-insert-workloads-part-1.aspx)

**Resolving PAGELATCH Contention on Highly Concurrent INSERT Workloads**

http://sqlcat.com/Themes/default/images/common/star-left-on.gifhttp://sqlcat.com/Themes/default/images/common/star-right-on.gifhttp://sqlcat.com/Themes/default/images/common/star-left-on.gifhttp://sqlcat.com/Themes/default/images/common/star-right-on.gifhttp://sqlcat.com/Themes/default/images/common/star-left-on.gifhttp://sqlcat.com/Themes/default/images/common/star-right-on.gifhttp://sqlcat.com/Themes/default/images/common/star-left-on.gifhttp://sqlcat.com/Themes/default/images/common/star-right-on.gifhttp://sqlcat.com/Themes/default/images/common/star-left-off.gifhttp://sqlcat.com/Themes/default/images/common/star-right-off.gif

**Authors: Thomas Kejser, Lindsey Allen, Arvind Rao and Michael Thomassy**

**Contributors and reviewers: Mike Ruthruff, Lubor Kollar, Prem Mehra, Burzin Patel, Michael Thomassy, Mark Souza, Sanjay Mishra, Peter Scharlock, Stuart Ozer, Kun Cheng and Howard Yin**

**Introduction**

Recently, we performed a lab test that had a large OLTP workload in the Microsoft Enterprise Engineering Center. The purpose of this lab was to take an intensive Microsoft SQL Server workload and see what happened when we scaled it up from 64 processors to 128 processors. (Note: This configuration is supported as part of the Microsoft SQL Server 2008 R2 release.). The workload had highly concurrent insert operations going to a few large tables.

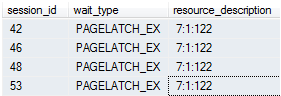
As we began to scale this workload up to 128 cores, the wait stats captured were dominated by PAGELATCH\_UP and PAGELATCH\_EX. The average wait times were tens of milliseconds, and there were a lot of waits. These waits were not expected, or they were expected to be a few milliseconds only.

In this TechNote we will describe how we first diagnosed the problem and how we then used table partitioning to work around it.

**Diagnosing the Problem**

When you see large waits for PAGELATCH in **sys.dm\_os\_wait\_stats**, you will want to do the following. Start your investigation with **sys.dm\_os\_waiting\_tasks** and locate a task waiting for PAGELATCH, like this:

SELECT session\_id, wait\_type, resource\_description   
FROM sys.dm\_os\_waiting\_tasks  
WHERE wait\_type LIKE 'PAGELATCH%'    
  
*Example Output:*



The **resource\_description** column lists the exact page being waited for in the format: **<database\_id>:<file\_id>:<page\_id>**.

Using the **resource\_description** column, you can now write this rather complex query that looks up all these waiting pages:

SELECT wt.session\_id, wt.wait\_type, wt.wait\_duration\_ms   
, s.name AS schema\_name   
, o.name AS object\_name   
, i.name AS index\_name   
FROM sys.dm\_os\_buffer\_descriptors bd   
JOIN (   
SELECT \*   
, CHARINDEX(':', resource\_description) AS file\_index   
, CHARINDEX(':', resource\_description  
  , CHARINDEX(':', resource\_description)) AS page\_index   
, resource\_description AS rd   
FROM sys.dm\_os\_waiting\_tasks wt   
WHERE wait\_type LIKE 'PAGELATCH%'   
) AS wt   
ON bd.database\_id = SUBSTRING(wt.rd, 0, wt.file\_index)   
AND bd.file\_id = SUBSTRING(wt.rd, wt.file\_index, wt.page\_index)   
AND bd.page\_id = SUBSTRING(wt.rd, wt.page\_index, LEN(wt.rd))  
JOIN sys.allocation\_units au ON bd.allocation\_unit\_id = au.allocation\_unit\_id  
JOIN sys.partitions p ON au.container\_id = p.partition\_id  
JOIN sys.indexes i ON p.index\_id = i.index\_id AND p.object\_id = i.object\_id  
JOIN sys.objects o ON i.object\_id = o.object\_id

JOIN sys.schemas s ON o.schema\_id = s.schema\_id

The query shows that the page we are waiting for is in a clustered index, enforcing the primary key, of a table with this structure:

CREATE TABLE HeavyInsert (   
  ID INT PRIMARY KEY CLUSTERED   
 , col1 VARCHAR(50)   
) ON [PRIMARY]

What is going on here, why are we waiting to access a data page in the index?

**Background Information**

To diagnose what was happening in our large OLTP workload, it’s important to understand how SQL Server handles the insertion of a new row into an index. When a new row is inserted into an index, SQL Server will use the following algorithm to execute the modification:

1. Record a log entry that row has been modified.
2. Traverse the B-tree to locate the correct page to hold the new record.
3. Latch the page with PAGELATCH\_EX, preventing others from modifying it.
4. Add the row to the page and, if needed, mark the page as dirty.
5. Unlatch the page.

Eventually, the page will also have to be flushed to disk by a checkpoint or lazy write operation.

However, what happens if all the inserted rows go to the same page? In that case, you can see a queue building up on that page. Even though a latch is a very lightweight semaphore, it can still be a contention point if the workload is highly concurrent. In this customer case, the first, and only, column in the index was a continuously increasing key. Because of this, every new insert went to the same page at the end of the B-tree, until that page was full. Workloads that use IDENTITY or other sequentially increasing value columns as primary keys may run into this same issue at high concurrency too.

**Solution**

Whenever many threads need synchronized access to a single resource, contention can occur. The solution is typically to create more of the contended resource. In this case, the contended resource is the last page in the B-tree.

One way to avoid contention on a single page is to choose a leading column in the index that is not continually increasing. However, this would have required an application change in the customer’s system. We had to look for a solution that could be implemented within in the database.

Remember that the contention point is a single page in a B-tree. If only there was a way to get more B-trees in the table. Fortunately, there IS a way to get this: Partition the table. The table can be partitioned in such a way that the new rows get spread over multiple partitions.

First, create the partition function and scheme:

CREATE PARTITION FUNCTION pf\_hash (TINYNT) AS RANGE LEFT FOR VALUES (0,1,2)   
  
CREATE PARTITION SCHEME ps\_hash AS PARTITION pf\_hash ALL TO ([PRIMARY])

This example uses four partitions. The number of partitions you need depends on the amount of INSERT activity happening on the table. There is a drawback to hash-partitioning the table like this: Whenever you select rows from the table, you have to touch all partitions. This means that you need to access more than one B-tree – you will not get partition elimination. There is a CPU cost and latency cost to this, so keep the number of partitions as small as possible (while still avoiding PAGELATCH). In our particular customer case, we had plenty of spare CPU cycles, so we could afford to sacrifice some time on SELECT statements, as long as it helped us increase the INSERT rate.

Second, you need a column to partition on, one that spreads the inserts over the four partitions. There was no column available in the table for this in the Microsoft Enterprise Engineering Center scenario. However, it is easy to create one. Taking advantage of the fact that the ID column is constantly increasing in increments of one, here is a simple hash function of the row:

CREATE TABLE HeavyInsert\_Hash(   
  ID INT NOT NULL   
  , col1 VARCHAR(50)   
  , HashID AS CAST(ABS(ID % 4) AS TINYINT)  PERSISTED NOT NULL)

With the **HashID** column, you can cycle the inserts between the four partitions. Create the clustering index in this way:

CREATE UNIQUE CLUSTERED INDEX CIX\_Hash   
ON HeavyInsert\_Hash (ID, HashID) ON ps\_hash(HashID)

By using this new, partitioned table instead of the original table, we managed to get rid of the PAGELATCH contention and increase the insertion rate, because we spread out the high concurrency across many pages and across several partitions, each having its own B-tree structure. We managed to increase the INSERT rate by 15 percent for this customer, with the PAGELATCH waits going away on the hot index in one table. But even then, we had CPU cycles to spare, so we could have optimized further by applying a similar trick to other table with high insert rates.

Strictly speaking, this optimization trick is a logical change in the primary key of the table. However, because the new key is just extended with the hash value of the original key, duplicates in the ID column are avoided.

The single column unique indexes on a table are typically the worst offender if you are experiencing PAGELATCH contention. But even if you eliminate this, there may be other, nonclustered indexes on the table that suffer from the same problem. Typically, the problem occurs with single column unique keys, where every insert ends up on the same page. If you have other indexes in the table that suffer from PAGELATCH contention, you can apply this partition trick to them too, using the same hash key as the primary key.

Not all applications can be modified, something that is a challenge for ISVs. However, if you DO have the option of modifying the queries in the system, you can add an additional filter to queries seeking on the primary key.

Example: To get partition elimination, change this:

      SELECT \* FROM HeavyInsert\_Hash   
      WHERE ID = 42

To this:

SELECT \* FROM HeavyInsert\_Hash   
      WHERE ID = 42 AND HashID = CAST(ABS(42 % 4) AS TINYINT)

With partition elimination, the hash partitioning trick is almost a free treat. You will still add one byte to each row of the clustered index.

Published [Sep 22 2009, 02:50 PM](http://sqlcat.com/technicalnotes/archive/2009/09/22/resolving-pagelatch-contention-on-highly-concurrent-insert-workloads-part-1.aspx) by [tkejser](http://sqlcat.com/members/tkejser.aspx)

Filed under: [Performance](http://sqlcat.com/technicalnotes/archive/tags/Performance/default.aspx), [OLTP](http://sqlcat.com/technicalnotes/archive/tags/OLTP/default.aspx)

**Comments**

|  |  |
| --- | --- |
| http://sqlcat.com/utility/anonymous.gif | [**The Rambling DBA: Jonathan Kehayias**](http://sqlblog.com/blogs/jonathan_kehayias/archive/2009/09/22/sqlcat-technical-note-resolving-pagelatch-contention-on-highly-concurrent-insert-workloads-part-1.aspx) **said:**  Its funny to see this new technical note posted by the SQLCAT team today because we just recently had  September 22, 2009 9:29 PM |
| http://sqlcat.com/utility/anonymous.gif | [**linchi\_shea**](http://sqlcat.com/user/Profile.aspx?UserID=3209) **said:**  I understand the changes you made improved the INSERT throughput. But what kind of throughput did you get when you ran the workload without any change on 128 cores as compared to running it on 64 cores? Was it worse on 128 cores than on 64 cores? Or was it better than on 64 cores, but wasn't as good as you had expected?  September 24, 2009 10:38 PM |
| http://sqlcat.com/users/avatar.aspx?userid=2218&lastmodified=633391322677589903 | [**tkejser**](http://sqlcat.com/user/Profile.aspx?UserID=2218) **said:**  Linchi: We did not have the opportunity to test a completely unchanged 64 core vs a fullly partitioned 128. You mileage will very depending on how many tables you have that need this trick. We have throughput increases up to 50% in some customer cases.I am currently lab testing a 32 core box with this trick and have so far gone from around 17K Sql tx/sex to 26 Sql tx/sec with this trick (and the box is not at 100% CPU load yet)  Note that removing PAGELATCH contention on one active table/index will often just expose another one in line to have the same optimization done.  September 25, 2009 3:12 AM |
| http://sqlcat.com/utility/anonymous.gif | [**linchi\_shea**](http://sqlcat.com/user/Profile.aspx?UserID=3209) **said:**  Thanks Thomas!  > We did not have the opportunity to test a completely unchanged 64  > core vs a fullly partitioned 128.  Absolutely no issue with the partitioning and hashing trick. Just want to make sure that we are not talking pass each other. I was curious about the INSERT throughput of the >unchanged< original workload on 64 cores vs the unchanged original workload on 128 cores. In particular, I'm curious about whether going from 64 cores to 128 cores in itself presented any issue to the original workload. Or, was it the case that the INSERT throughput did increase with the unchanged workloadwas, bu the increase was much smaller than what you had expected?  I know the increased PAGELATCH waits caught your attention as you scaled to 128 cores. But as much as one would want to reduce the PAGELATCH waits, the waits were really secondary as the real measure of interest to the customer was the throughput.  September 25, 2009 11:53 AM |
| http://sqlcat.com/utility/anonymous.gif | [**Log Buffer #163: a Carnival of the Vanities for DBAs | Pythian Group Blog**](http://www.pythian.com/news/4118/log-buffer-163-a-carnival-of-the-vanities-for-dbas) **said:**  Pingback from  Log Buffer #163: a Carnival of the Vanities for DBAs | Pythian Group Blog  September 25, 2009 1:05 PM |
| http://sqlcat.com/utility/anonymous.gif | [**Log Buffer #163: a Carnival of the Vanities for DBAs « PlanetMysql.ru – ???????????????????? ?? ???????? MySQL**](http://planetmysql.ru/2009/09/25/log-buffer-163-a-carnival-of-the-vanities-for-dbas/) **said:**  Pingback from  Log Buffer #163: a Carnival of the Vanities for DBAs &laquo;  PlanetMysql.ru &#8211; ???????????????????? ?? ???????? MySQL  September 25, 2009 3:25 PM |
| http://sqlcat.com/utility/anonymous.gif | [**Alexander Gladchenko**](http://msmvps.com/blogs/gladchenko/archive/2009/09/28/1727878.aspx) **said:**  div align=justify&gt; По материалам статьи: &quot; Resolving PAGELATCH Contention on Highly Concurrent  September 28, 2009 8:32 AM |
| http://sqlcat.com/utility/anonymous.gif | [**Russian SQL Server Club**](http://www.itcommunity.ru/blogs/rsug/archive/2009/09/28/78230.aspx) **said:**  По материалам статьи: "Resolving PAGELATCH Contention on Highly Concurrent INSERT Workloads".  ...  September 28, 2009 8:44 AM |
| http://sqlcat.com/users/avatar.aspx?userid=2218&lastmodified=633391322677589903 | [**tkejser**](http://sqlcat.com/user/Profile.aspx?UserID=2218) **said:**  Linchi: Unfortunately, we did not get to do a 1-1 compare. One of the reasons was the we hit the PAGELATCH wait BEFORE reaching 64 cores - so we had to apply these tricks already then.  Last week, I did a lab on a 32 core box and before we reached 50% CPU load we had to apply this trick to go higher on throughput (which is now around 30K SQL tx/sec). There was a very hot table in that workload which received at least one insert with every transaction in the system.  October 1, 2009 3:48 AM |
| http://sqlcat.com/utility/anonymous.gif | [**dpetrancuri**](http://sqlcat.com/user/Profile.aspx?UserID=3238) **said:**  I have a table with +2 bil records where the table has a clustered index (thus cannot be minimally logged insert) where the new records are (by multiple orders of magnitude) most always inserted on the last pages at the end of the B-tree.  So, in reading this, if I have a massively large insert (millions of records per statement) against a table with a similiar index structure, the question comes to mind...Could insert performance within a single statement be adversely affected because of PAGELATCHing when the query is 'parallelized' and the table is not partitioned (in a SQL Server 2000 environment)?  I eagerly await your reply  October 6, 2009 1:19 PM |
| http://sqlcat.com/users/avatar.aspx?userid=2218&lastmodified=633391322677589903 | [**tkejser**](http://sqlcat.com/user/Profile.aspx?UserID=2218) **said:**  Dpetracuri: There is a limit to how fast you can insert data with a single statement. Note that the insert part of the query plan is single threaded. This also means that there is no need for thread coordination and hence no need for latches to be coordinated.  But if you had two or more insert statements, each adding many rows at same time - you could see this earlier than in the case we had in the lab.  October 6, 2009 1:29 PM |
| http://sqlcat.com/utility/anonymous.gif | [**dpetrancuri**](http://sqlcat.com/user/Profile.aspx?UserID=3238) **said:**  tkejser: is that statement about single threading also true of partitioned tables with an aligned partitioned index or does parallelization take place during inserts to partitioned tables because of the true physical separation of the data and indexes?  Thank you for your learned reply.  October 8, 2009 12:54 PM |
| http://sqlcat.com/users/avatar.aspx?userid=2218&lastmodified=633391322677589903 | [**tkejser**](http://sqlcat.com/user/Profile.aspx?UserID=2218) **said:**  Dpetrancuri: Yes, it is also true for a partitioned table. There is still a single thread feeing the rows to the insert operator - even though it hits multiple partitions.  But you have the right idea.  Howver, with a partitioned table you can switch out the individual partition and run one INSERT statement per partition. After that, you can switch the partitoins back in again. Actually I just did this on a 32 core box today. With 32 (switched out) partitions the aggregate insert throughput was around 1GB/sec (heap tables, using the new bulk logged INSERT in SQL 2008). Even under that speed, i did not run into the PAGELATCH, because each statement touched it's own partition.  October 8, 2009 1:02 PM |
| http://sqlcat.com/users/avatar.aspx?userid=3682&lastmodified=634285252615695991 | [**Paul White NZ**](http://sqlcat.com/user/Profile.aspx?UserID=3682) **said:**  Since partitioning is an Enterprise-only feature, I think it's worth pointing out that a similar effect can be produced using a local partitioned view, though it does require INSTEAD OF triggers on the view.  USE tempdb;  GO  CREATE  TABLE         dbo.HeavyInsert1         (         id      INTEGER NOT NULL,         ptn     TINYINT NOT NULL                 CHECK (ptn = 1),         col1    VARCHAR(50) NOT NULL,         PRIMARY KEY (id, ptn)         );  GO  CREATE  TABLE         dbo.HeavyInsert2         (         id      INTEGER NOT NULL,         ptn     TINYINT NOT NULL                 CHECK (ptn = 2),         col1    VARCHAR(50) NOT NULL,         PRIMARY KEY (id, ptn)         );  GO  CREATE  VIEW         dbo.HeavyInsert  WITH    SCHEMABINDING  AS  SELECT  id, ptn, col1  FROM    dbo.HeavyInsert1  UNION   ALL  SELECT  id, ptn, col1  FROM    dbo.HeavyInsert2  GO  CREATE  TRIGGER [trg dbo.HeavyInsert IOI]  ON      dbo.HeavyInsert  INSTEAD OF INSERT  AS  BEGIN     SET NOCOUNT ON     INSERT dbo.HeavyInsert (id, ptn, col1)     SELECT id, id % 2 + 1, col1 FROM inserted  END  GO  INSERT  dbo.HeavyInsert (id, col1) VALUES (1, 'TestRecord1')  INSERT  dbo.HeavyInsert (id, col1) VALUES (2, 'TestRecord2')  INSERT  dbo.HeavyInsert (id, col1) VALUES (3, 'TestRecord3')  INSERT  dbo.HeavyInsert (id, col1) VALUES (4, 'TestRecord4')  GO  SELECT  \*  FROM    dbo.HeavyInsert  WHERE   id = 4  AND     ptn = (4 % 2 + 1);  GO  DROP    VIEW dbo.HeavyInsert;  DROP    TABLE dbo.HeavyInsert1;  DROP    TABLE dbo.HeavyInsert2;  May 2, 2010 1:42 AM |