**[Technical Notes](http://sqlcat.com/technicalnotes/default.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.

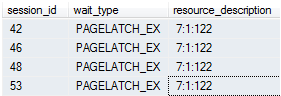
As we began to scale this workload up to 128 cores, the wait stats captured were dominated by PAGELATCH\_UP and PAGELATCH\_EX. The average wait times were tens of milliseconds, and there were a lot of waits. These waits were not expected, or they were expected to be a few milliseconds only.

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.

Latches are lightweight internal structures used by SQL Server database engine to ensure physical data integrity. The large majority of latches are acquired when a data page is moved from the storage engine to the data cache but there are latches that are used for other types of synchronizations. Each latch is associated with a single allocation unit. Locks and latches are similar objects but they have a slightly different purpose. Locks ensure that the same data element cannot be modified by two different connections at the same time; latches, on the other hand, simply ensure that the data page on which the data element resides is physically readable and writable. A latch is only maintained on the data page while the data is being changed, as opposed to locks which are maintained for the duration of the entire transaction.   
  
  
  
Each data page in the buffer cache is associated with a structure that tracks the status information, including whether the page has been modified or not (whether it is "dirty"). A data page can only be latched while it is in memory, as opposed to locks which can be held on pages that aren't in the buffer cache.

After a data page is modified in memory it must be written to disk. Before writing the data page to disk SQL Server latches it to prevent further modifications. Once the page is successfully written to disk, the latch is released.   
  
  
  
Latching is also used when the data is read. SQL Server retrieves data pages that are likely to be used and places them in the buffer cache. This functionality is called "read-ahead". During read-ahead, SQL Server acquires latches on data pages to protect them from further access and from modification. Once the pages have been moved to the buffer cache, SQL Server double-checks each page for a valid page number and torn page errors. After these checks are complete the latches are released.   
  
  
  
Latch contention can occur if a request for a latch cannot be granted immediately. This typically happens because the latch is held by another thread.