



We [SOLVE COMPLEX PROBLEMS](#) of [DATA MODELING](#) and [DEVELOP TOOLS](#) and solutions to let business perform best through data analysis

Whitepaper: performance of SqlBulkCopy

This whitepaper provides an analysis of the performance of the bulk loading of huge tables inside SQL 2008 using .NET code, SSIS and various options.

From .NET 2.0 Microsoft has released the SqlBulkCopy class, which is an interface between the BULK INSERT command in T-SQL and .NET applications. SqlBulkCopy has some properties that might be used to change its behavior and is able to load data from DataRow arrays, DataTables or IDataReaders.

In this paper we will provide some hints for the usage of the SqlBulkCopy class and provide a complete implementation of a multithreaded environment that leverages this class to obtain high speed bulk loading with SQL Server.

We are not going to explain the obvious details of the class nor anything that is already well covered in the books online, the goal of this paper is to describe the best techniques to adopt in order to make the process of bulk loading as fast as we can.

In the first part of the paper we will provide a brief description of the overall architecture of the test program, then we will describe some tests, with the aim to find the best configuration for the whole architecture and the various parameters that can be changed in the SqlBulkCopy class.

Since this paper is an investigative one, it does not contain sentences like “This is the truth” but more like “this is what we have measured and tested, trying to understand what is going on under the cover”. It might be the case that some considerations are not perfectly correct... nevertheless, this is what we discovered.

What the reader should be familiar with:

- A solid knowledge of the .NET environment
- Basic concepts of parallel programming in the .NET environment
- A good understanding of SQL Server internals is welcome

Please note that a very good explanation of all the results described here (and a lot more) can be found in this great whitepaper: <http://msdn.microsoft.com/en-us/library/dd425070.aspx>, which contains detailed explanation on how to reach the best bulk insert operations. Nevertheless, the whitepaper is pretty hard to read and understand and describes some exoteric hardware that is pretty difficult to find in the real world.

We find that this documents can be of help to anybody wanting to improve bulk insert operation without the need to go deep inside the bits as the Microsoft whitepaper does.

That said, let us start with the interesting stuff!

The Producer / Consumer pattern

The classic way to load data from a source to a destination is that of producing some rows in a pattern like this:

- The producer reads data from a specified source
- It operates some transformation on the data
- It then writes them in the final form to the database (in general, to the consumer)

This is something normally done with SSIS for large volumes of data and simple transformations. Nevertheless, there is sometimes the need to create .NET code when the transformation process is very complex and SSIS is not the best suitable solution. Yet, the need to handle huge volumes of data lead us to the need to search for the best performances.

In order to get the best from modern multi-core computers, we can write many producers that handle the data in parallel and send them to many consumers, so that all the operations of read / transform / write can be scaled on many cores. Moreover, we would not be limited to a direct link between one consumer one producer: we can have, for example, 10 producers and 5 consumers, depending on which process is the slowest and, in consequence, needs more horsepower.

The `SqlBulkCopy` class is not very well suited for such an architecture, since we can only call the `WriteToServer` method sending it a bulk of data (a `DataTable` or a `DataReader`, no difference on it, but just one). Calling it multiple times creates the necessity to handle multiple transaction. Thus, if we have 10 producers, we need to serialize them on a single consumer or create 10 consumers. It is time to think that we need to write some code to override this limitation.

The test program we wanted to use has a slightly complex architecture where:

- A producer is tied to a single consumer. When the producer generates some data, it places them into the buffer of its consumer.
- A consumer can handle multiple producers and, when writing data to the server, it will wait until all its producers have finished their work before closing the transaction.
- If a consumer has no data available, it will wait for some producers to generate it, entering sleep mode.

In order for the consumer to manage this operation, it uses a buffer that implements `IDataReader` and handles all the complex operations needed. `SqlBulkCopy` will read data from the buffer thinking it is a `DataReader` and, if the buffer detects that no data is yet available, it will pause the operation until some data comes in.

Using this architecture, we can have as many producers sending data to multiple consumers. All threads can work in parallel until the whole operation is finished, using as many cores as we need. Moreover, if both producers and consumers will work at the same time, the memory footprint will be reduced since data is flushed to the database as soon as it is produced and no cache is needed.

It is pretty clear that, having the opportunity to launch more than one bulk insert on the same table in parallel, there is now the problem whether this opportunity is useful or not and, if the answer is yes, how many parallel insert can we afford before degrading performances? More on this later. ☺

What we tested

The interesting parameters in the SqlBulkCopy class are:

- **Table locking**. During the bulk insert we can put an exclusive lock on the table, this speeds up the insert operation, the goal is to measure it in various scenarios.
- **BatchSize**: this parameter defines the size of each single batch that is sent to the server. There is no clear documentation about the effect of this parameter on performances.
- **Use Internal Transaction**. Honestly, we had problems in understanding the behavior of SqlBulkCopy when it comes to transaction handling. We'll spend some time in investigating which options we have available for transactions. That said, it is pretty normal to perform bulk insert with no transaction handling at all, since we expect it to be faster.

Moreover, there are other interesting topics to investigate about bulk loading, like:

- We can load both clustered indexed tables and heaps. Which one is faster and what can we do to load data in the best way?
- What about the usage of the log file? Bulk operations should be minimally logged but it is very interesting to see exactly how much log space is used and whether there are interactions between the various settings and the log size.
- SQL Server Trace Flag 610 provides minimal logging for bulk insert on clustered tables. Is it worth using or not?

How we tested

To perform tests we created a simple table with six fields and a row size of approximately 320 bytes.

```
CREATE TABLE dbo.Test (  
    FieldChar1 VARCHAR(100) NOT NULL,  
    FieldChar2 VARCHAR(100) NOT NULL,  
    FieldChar3 VARCHAR(100) NOT NULL,  
    FieldInt1 INT NOT NULL,  
    FieldInt2 INT NOT NULL,  
    FieldInt3 INT NOT NULL)
```

We want to fill it with some records, starting from 6 millions up to 60, in order to see if the growth in time is linear or not.

Moreover, in order to test the table as both a heap and a clustered table, we alternatively used this index:

```
CREATE CLUSTERED INDEX TestIndex ON Test (FieldInt1, FieldInt2, FieldInt3)
```

The test table is destroyed and recreated each time on a database that already has enough space to contain it, so that we do not need to make it grow during the bulk copy operation. The data inserted in the table is made up of random values, computed each time differently. We made two different tests with the random values:

- Each producer generates all its rows containing exactly the same value

- Each producer generates each row as a random one, creating a completely unsorted and messed up data table.

The goal of this test is to determine if there is some form of compression in the data sent to the network (the same value repeated a lot of time should give a good level of compression, while a messed up dataset will generate larger network traffic).

The database is in Simple logging mode, since this is the logging option mostly used in the data warehouse field, where SqlBulkCopy is mostly used.

We did not test all the different combinations of options in the SqlBulkCopy class, since we are not interested in a complete diagram but in the analysis of the effect that each option has on the overall performance of bulk loading.

In order to perform correct measurements, producers and consumers have been sometimes serialized. This means that, before starting the first SqlBulkCopy operation, we wait for all the producers to generate rows. This is necessary in order to be sure that the producers have enough horsepower to complete their job. After all, we are interested in the performance of SqlBulkCopy, not in those of the producers.

How much horsepower for SqlBulkCopy?

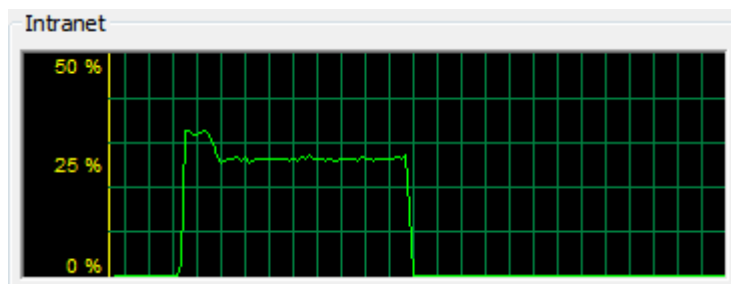
The first test performed is the most simple one: one consumer (i.e. one instance of SqlBulkCopy running) receives 6 million rows and send them on one heap. The number of producers has no influence since the producer and consumer processes are serialized.

The consumer execution time is 56 seconds and we can see that using Tablock and a value of BatchSize equal to zero we get minimal logging (6 Mb of log file is very small for a 6 million rows table)

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 24
Number of consumers: 1
Number of rows      : 6.000.000
Batch size         : 0
Table Locking      : Activated
Transaction        : Disabled

Produce data       : 6.853 milliseconds
Consume data       : 56.494 milliseconds
Log file usage     : 6,23 Mb
```

The interesting point to note is the network usage during the operation.



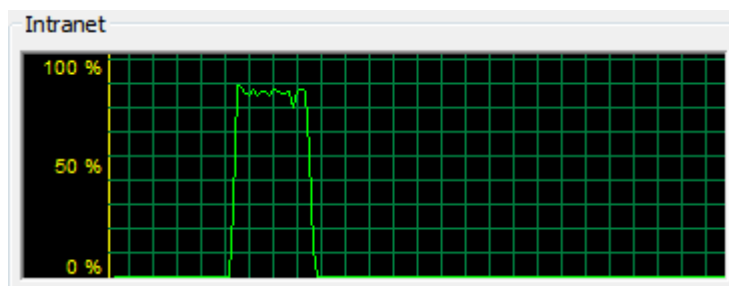
We can see that the network usage (which has a bandwidth of 1Gbit) is around 25%. The CPU view shows that only one thread is running and it does not consume a whole core. So, it seems that – having more than one core on the client – we can increase the overall speed by adding more than one SqlBulkCopy operation in parallel, since we have plenty of space to improve performance.

A very simple computation tells us that, if a single consumer uses 25% of the network bandwidth, we can run four in parallel trying to use the whole bandwidth. Here are the results:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 24
Number of consumers: 4
Number of rows      : 6.000.000
Batch size         : 0
Table Locking      : Activated
Transaction        : Disabled

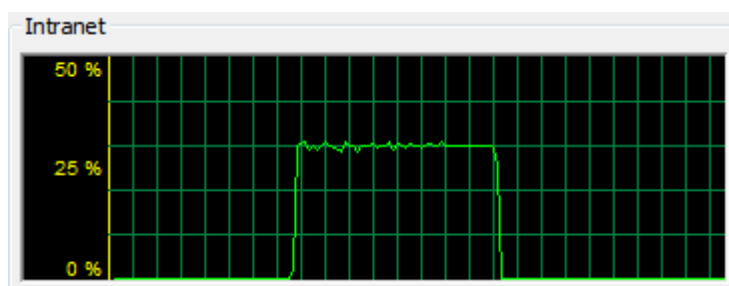
Produce data       : 6.147 milliseconds
Consume data       : 19.322 milliseconds
Log file usage     : 6,26 Mb
```

Running four SqlBulkCopy operation in parallel produces the same results as before but runs in less than 20 seconds, which is a 300% increase in speed! Looking at the network usage we see that now the full bandwidth is used for a smaller time:

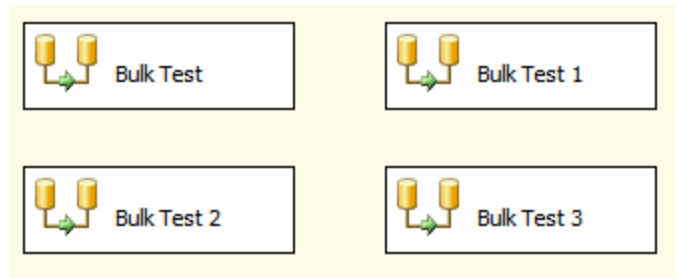


.NET versus SSIS, which is the fastest?

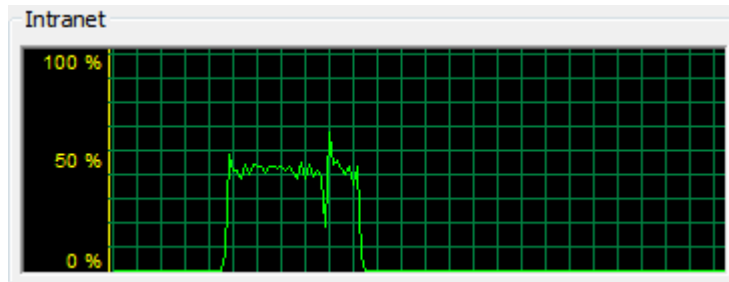
At this point, it might be interesting to see what happens if we run the same test using an SSIS package to bulk load data into the table. To perform this test we created a very simple package that uses a script source component to generate data and simply sends all data to a destination that uses fastload on the table. The execution time of the package is around 52 seconds (which includes both production and consumption of data) but, the interesting point is the network usage:



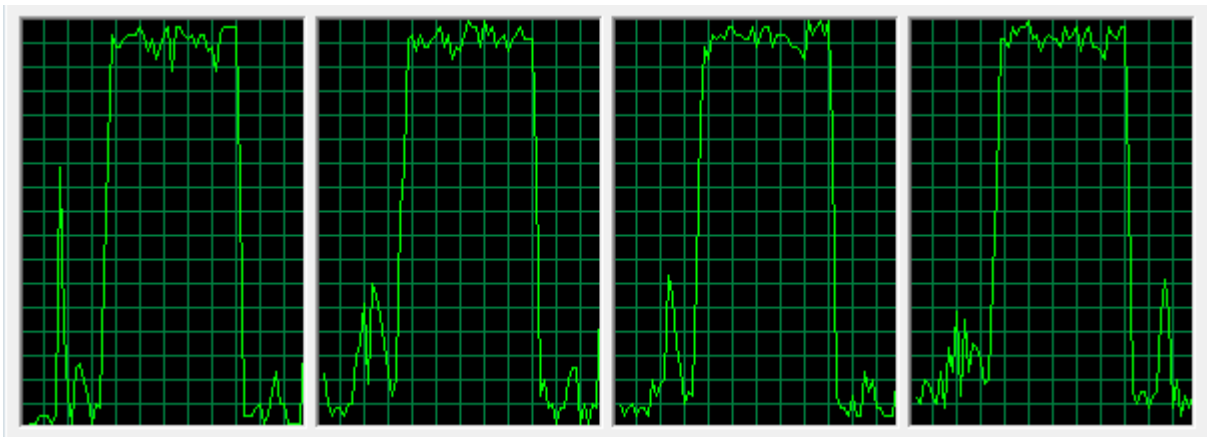
We can see that it runs at about 30% all the time during the loading process. What can we learn from this? The bulk loading inside SSIS 2008 seems faster than the same operation done with .NET since – we guess – there are few operations done to perform type validation and conversion. Nevertheless, the network usage is far beyond the 100%, meaning that the insert process is done in a single threaded component which is not able to fill the client and server resources. Clearly, using SSIS we normally perform many operations in parallel, yet we have to remember that, for a single very large fact table, we can have a significant gain in speed if we load it using more than one process. In order to better test it, we changed the package structure to have four data flow tasks running in parallel:



Surprisingly, we get an execution time of 35 seconds and the network usage is far from optimal:



We can see that the network bandwidth is used at around 50%, half than what we can get with the .NET code. Well... it is true that this test runs both the producer and consumer at the same time, in fact – looking at the CPU usage – we see that all four cores are running at top speed for the whole time:



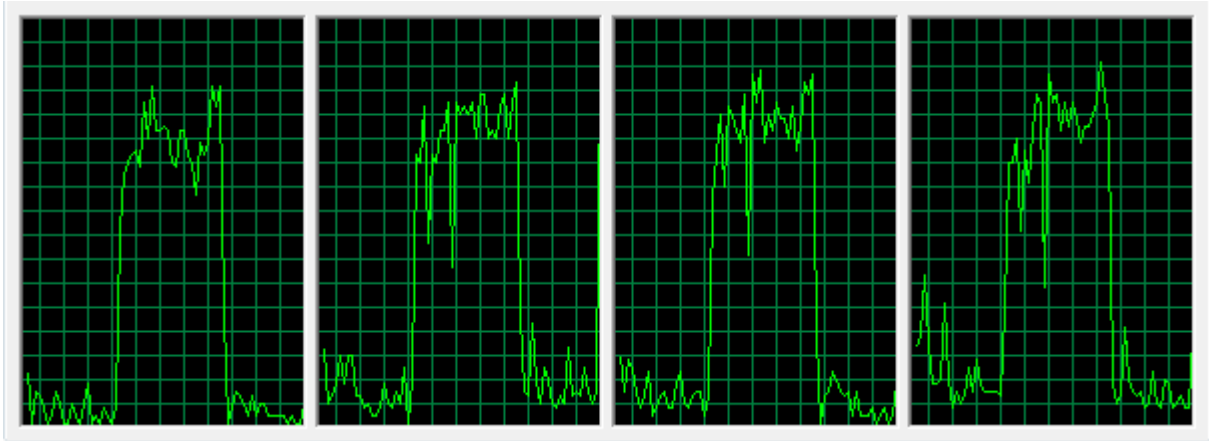
In this case, it is possible that the number of cores is not enough to provide horsepower to both the producer and consumer processes.

It is now time to try the .NET program running both producer and consumer at the same time, and take a look at what happens there. In order to have the same configuration, we provided 4 producer and 4 consumers to run at the same time. The time shown is pertinent to both producers and consumer running in parallel and it is very interesting to note that it is around 26 seconds, much faster than the SSIS package and very near to the sum of produce and consume processes, when run serialized:

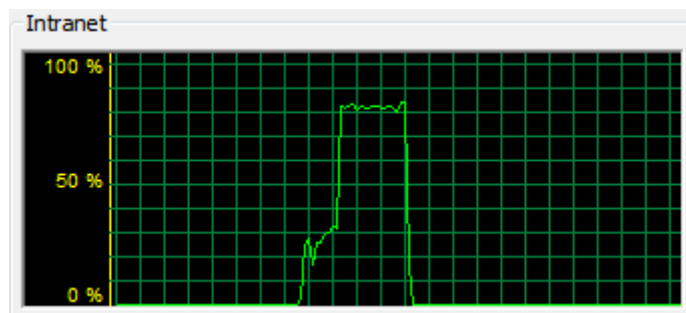
```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
```

```
Number of producers: 4  
Number of consumers: 4  
Number of rows      : 6.000.000  
Batch size          : 0  
Table Locking       : Activated  
Transaction         : Disabled  
  
Global time         : 25.872 milliseconds  
Log file usage      : 6,26 Mb
```

The CPU usage of the .NET code is this:



While the network usage seems interesting too:



We can see that 4 cores are able to handle 8 processes running in parallel (4 producers, 4 consumers) without ever reaching the CPU limits. Moreover, the network bandwidth, after an initial step where producers have not yet filled the buffers, rapidly reaches an 80% value and stays there up to the end of the process.

This means that the overhead introduced by the buffer management of SSIS is pretty heavy. We are not saying that SSIS is slow but that there seems to be very large areas of improvement since the same operation, done with .NET code, is much faster.

We are not going deeper in test with SSIS. We do not think that it is fair to compare only raw speed, there are a high number of very good reasons to use SSIS instead of .NET code to load data warehouses, nevertheless it is good to know that, if we need to get top speed for a specific phase of the ETL, then it might be the case to consider .NET coding for that phase and replace SSIS, although we will never consider .NET for a full data warehouse ETL process.

Reaching the number of consumers limit

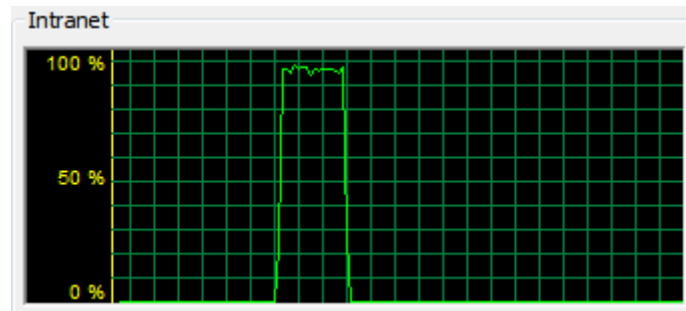
The next test, after having discovered that 4 consumers can run in parallel, is to determine the highest number of concurrent consumer that can run in parallel before we reach the point at which adding more processes is useless.

We tested 4, 6, 8 and 12 processes and found that the best value (at least in our test configuration) is to provide 8 consumers (that is, twice the number of cores). The network usage stays at 92/93% all the time and we get a very good value of 17 seconds to load 6 million rows:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 24
Number of consumers: 8
Number of rows      : 6.000.000
Batch size         : 0
Table Locking      : Activated
Transaction        : Disabled

Produce data       : 6.399 milliseconds
Consume data       : 16.937 milliseconds
Log file usage     : 6,27 Mb
```

The network usage is very good and we do not see any space for improvement (reaching the theoretical value of 100% is more a dream than a realistic goal):



Nevertheless, this is an hardware limit and, for specific configurations where the network bandwidth is larger than ours, it might be possible to increase that number. A realistic guess is that using two SqlBulkCopy for each core in the client computer leads to best performance, if we have a network connection fast enough to provide data to the server. Moreover, we will soon reach the hardware limits on the server, where – even with a very fast network – the server disks will not be able to load all that stuff fast enough.

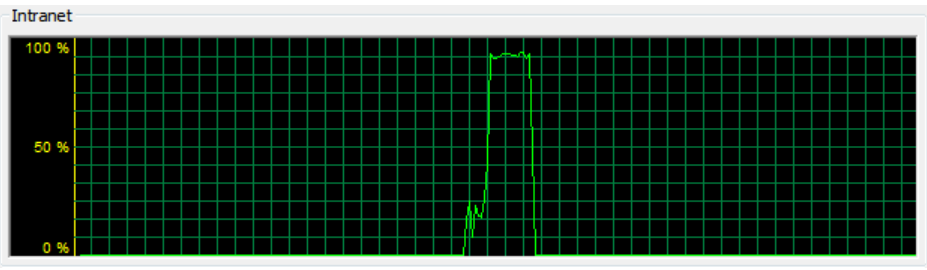
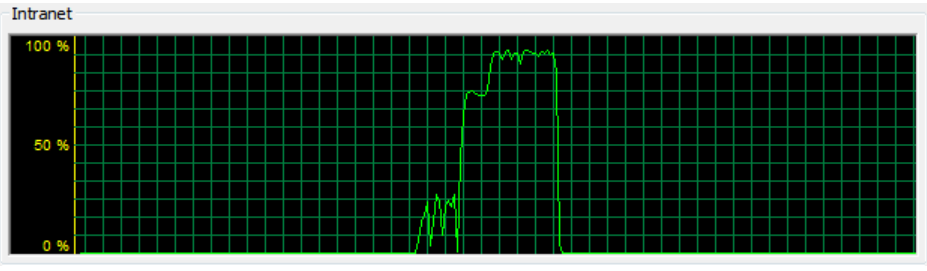
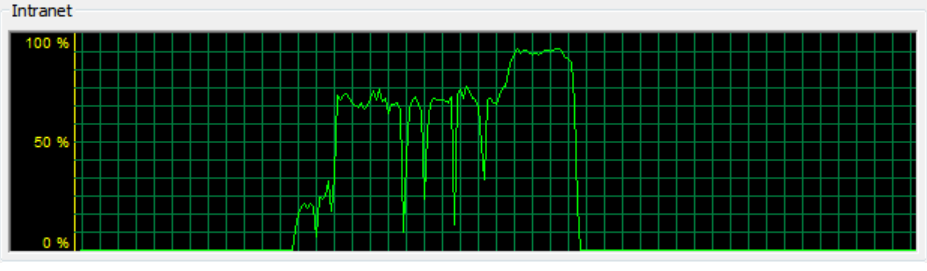
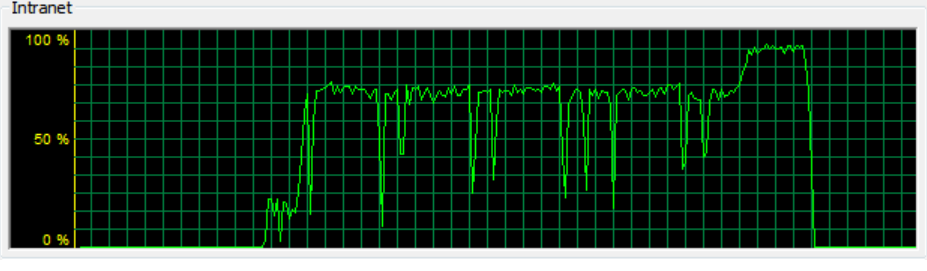
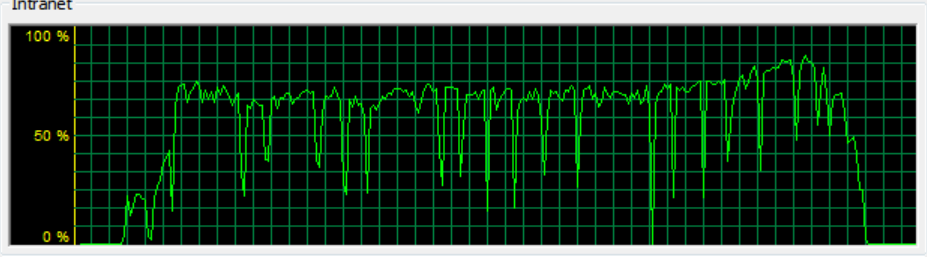
The important thing to note is that the limit is an hardware one, adding faster hardware will lead to faster ETL. The final tests should be carried on the customer's hardware, in order to get the best out of it. Needless to say, if we are developing an application, we should not hardwire the number of parallel operations since, changing the hardware, it might be necessary to change the degree of parallelism.

Increasing the number of rows

Now we have a test environment with 8 consumers, the next step is to look at what happens when the number of rows increases from the base of 6 millions up to 60 millions. In order to make the tests we created a 60 million rows table and then tested the various sizes truncating the table before loading it.

Please note that, since we cannot afford to hold 60 million of rows in memory (roughly 20Gb), these tests are run making both producers and consumers run in parallel. The producers will enter a wait status if the size of the buffer of their consumer is growing too big, thus reducing memory pressure.

In the following table we provide the results for various sizes of the table:

Rows	Time (sec)	Network usage graph
6	23.3	
12	47.6	
24	95.9	
48	184.9	
60	248.3	

It is very interesting to note that the network usage (and, consequently, the performance of the loading operation) is composed of

- An initial section, where the network traffic increases due to the producer starting to fill the buffers

- a middle section where there is a strong thread contention between producer and consumers (on the client side, all the 4 cores are running at 85/90%) where the network traffic stays around 75/80% and shows some points where it slows down to less than 20%.
- A final section, where the network traffic increases to 90% just before the final rush, where threads finish their work.

It is pretty easy to understand what is going on. The client does not have enough cores to handle in parallel both producer and consumers and so, during the whole time where producers are asking for CPU to generate data, the bulk loading does not work at its whole speed. When producer threads start to close and leave all the CPU power to consumers, then we see the bandwidth increase, since the operating system is free to leave consumer processes to have all the CPU they need.

For what concerns the points in the graph where we see a sudden drop in the network bandwidth, we guess it is SQL Server flushing buffers to disk, thus making clients wait until its internal operation has completed.

In order to confirm that this is the case, we reduced the number of concurrent threads and found that, to load 60 million of rows, reducing to 6 consumers and 6 producers provided a gain of 11 seconds, lowering the overall time to 237 seconds against the 248 of 8 threads.

Anyway, if we need to draw some conclusion from this test, we can easily say that the growth in time is perfectly linear with the number of rows inserted, and this is somehow expected. What was not expected is the fact that if the number of lines to add increases, then the time during which there is CPU contention grows and so it is better to lower the number of threads in order to get best performances.

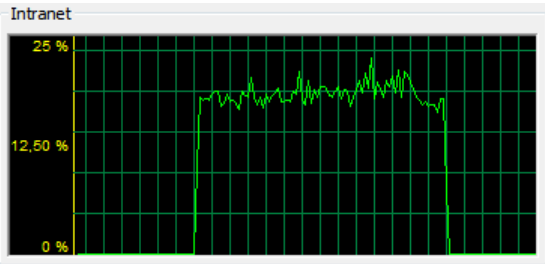
Setting some of the SqlBulkCopy options

Now it is time to play a bit with the various options of the SqlBulkCopy class.

Batch Size

The first interesting option is the BatchSize parameter. This parameter imposes a limit to the number of rows that should be sent in a single batch to SQL Server.

In order to test it we go back to the 6 million table with 8 producers and 8 consumers serialized, to get a better test of the consumers only. Then, we measured various timings with different values for the BatchSize parameter: 500, 1000, 10.000 and 100.0000. Moreover, since its value varies a lot, we recorded the log size.

Batch Size	Time (sec)	Log Size	Network usage graph
500	83.5	113 Mb	 <p>The graph, titled 'Intranet', shows network usage percentage over time. The y-axis ranges from 0% to 25% with major ticks at 0%, 12.50%, and 25%. The x-axis represents time. The usage starts at 0%, rises sharply to a plateau between 15% and 20%, remains relatively stable with some fluctuations, and then drops sharply back to 0% at the end of the period.</p>

Batch Size	Time (sec)	Log Size	Network usage graph
1.000	28.2	141 Mb	
10.000	16.98	20 Mb	
100.000	16.80	6 Mb	
0	16.7	6 Mb	

The results are quite surprising. If we use the BatchSize parameter, performance get worse and worse as we set it to lower values.

Using a very small value for the batch size the network bandwidth stays well under 20% of usage for the whole process (please note that the 500 batch size graph upper limit is 25%, which is different from the other graphs). Any value below 10.000 decreases performance in a very heavy way, leading to very poor time and a heavy log file usage.

When we reach 10.000 of Batch Size, then the difference in time among the various tests becomes very small. But, since we have 8 threads writing 750.000 rows each, then we only have 75 chunks of data sent from each thread. Needless to say, the best performance is obtained when we use 0 as BatchSize, sending the whole 750.000 rows in a single batch

It might be interesting to discover if there is any benefit, apart from performance, by using the BatchSize parameter. We did not find anything mentioned in the Books On Line nor have we ever seen anything interesting during our experience, this lead us to say that the best thing to do with BatchSize is to leave it to

zero, which is its default value, since any value different from that will decrease the performance of the load process.

Table Locking

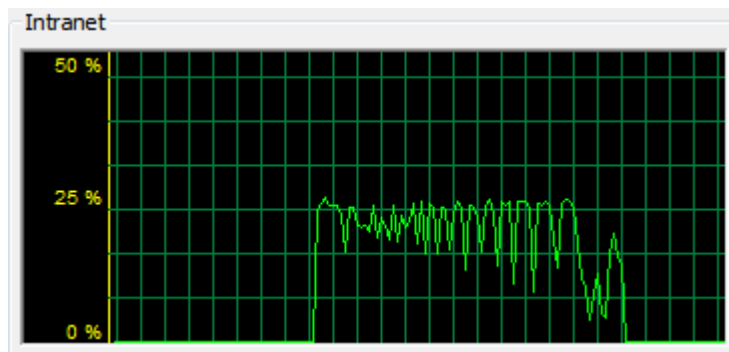
We expect table locking to have a significant impact on the performance of the bulk load operation. The only question is “how much will it affect performance”? All the tests up to now have been carried on with the table lock option on (which is not the default behavior).

If we remove the table lock option from the SqlBulkCopy operation, we obtain this result:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 8
Number of consumers: 8
Number of rows      : 6.000.000
Batch size         : 0
Table Locking      : Disabled
Transaction        : Disabled

Produce data       : 6.596 milliseconds
Consume data       : 69.566 milliseconds
Log file usage     : 2.429,25 Mb
```

And the network usage graph is the following:



We can see that the network usage is very low and the amount of data logged is huge, corresponding to the size of the table being loaded. This is definitely what we expected, since TABLOCK is needed to get minimal logged operations. Nevertheless, it would have been better to have TABLOCK set by default, in order to get best speed BULK INSERT operations by default.

Moreover, it is worth noting that the locks imposed by each thread do not interfere each other. So it is possible to perform many BULK INSERT operations in parallel, each one imposing a TABLOCK lock on the table but running them all in parallel, which is something we already knew from books on line. Nevertheless... stay tuned, there will be some amazing news about locking later on. ☺

Internal Transaction

The SqlBulkCopy class gives you an option to activate an internal transaction. It is not very clear what this means in term of transaction handling by the BULK INSERT operation, since we expect the SqlBulkCopy class to create and handle a transaction internally if we do not provide an external handling of that.

We performed many tests with and without the UseInternalTransaction setting and noticed no significant difference between the various tests.

It would have been interesting to provide an external transaction, in order to run the various consumers in parallel and inside a transaction started from the outside, but we did not succeed in making it run since – in

this case – all operations need to share the same connection and it seems that more than one BULK INSERT operation cannot be started on the same connection, even if we enable MARS.

Is the network traffic compressed?

In order to effectively test for a possible compression of data in the network, we reduced to 1 producer and 1 consumer and tested two different situations:

- The producer generates all identical rows.
- The producer generates all rows with random data and each one different from the preceding ones.

If there is some sort of compression, we expect the network traffic to be reduced when all the rows are identical. We needed to reduce the number of generated rows in order not to fill memory and get wrong results due to paging and garbage collection.

When all the rows are identical this is the result:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 1
Number of consumers: 1
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Activated
Transaction         : Disabled

Produce data        : 9.603 milliseconds
Consume data        : 37.038 milliseconds
Log file usage      : 4,16 Mb
```

While, if all the rows are different, the result is:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 1
Number of consumers: 1
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Activated
Transaction         : Disabled

Produce data        : 51.261 milliseconds
Consume data        : 37.038 milliseconds
Log file usage      : 4,16 Mb
```

We can see that there is no difference at all in the consume time. The big difference is in the producing of data (since the random generator must be called for each row). This led us to believe that there is no compression of data at the transport level and hence, from the network traffic point of view, providing data in a sorted or unsorted way is not important for the network performance.

As we will see, sorting of data leads to very different performance if we use a clustered table but, for what concerns a heap, there is no difference.

Bulk loading of a clustered table

All tests performed up to now have been carried on with a heap. It is intuitive to think that performance, for a clustered table, will be different. So, this is the next set of tests we wanted to carry on, in order to detect what affects the loading of clustered tables.

We created this clustered index:

```
CREATE CLUSTERED INDEX TestIndex ON Test (FieldInt1, FieldInt2, FieldInt3)
```

Since the clustered table is empty, we expect the loading from a single thread to be minimally logged. What we do not know is what will happen when many producers will insert data in the clustered table in parallel.

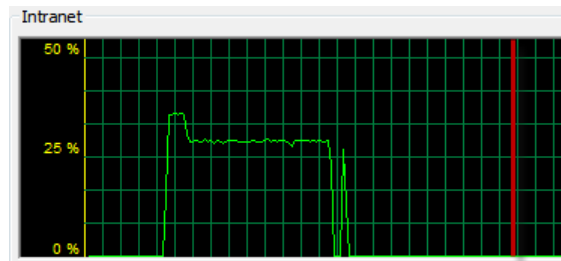
The first test is to use one producer, one consumer and load the clustered table. All the rows contain the same key. The result is:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM

Number of producers: 1
Number of consumers: 1
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Activated
Transaction         : Disabled

Produce data        : 9.170 milliseconds
Consume data        : 79.633 milliseconds
Log file usage      : 4,17 Mb
```

While the network usage graphs is interesting:



The program stopped at the red line, long after the network traffic was finished. This is the standard behavior when loading a clustered table since data, before being loaded, needs to be sorted. During the sorting the client is not working and the network is idle, the BULK INSERT operation is waiting for the final commit to be concluded before returning to the caller.

Anyway, it is interesting to note that the loading time of a clustered table is roughly 4 times the loading of a heap. Moreover, it is interesting to note that the creation of the index on the table spends 7 seconds. So, if we add 37 seconds (loading of the heap) to 7 seconds (creation of the clustered index), we get 44 seconds, which is half the time of loading the table with the clustered index active. It seems interesting to point out that, if we need to load a clustered indexed table, it might be useful to drop and re-create the clustered index after the loading. Obviously, there are a lot of other topics that must be addressed before taking such a decision, nevertheless it surely is worth investigation.

Before continuing with other tests, we need to note that the rows inserted contained the same key. What is going to happen when all the keys are generated randomly? Here are the results:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
```

```
Number of producers: 1
Number of consumers: 1
Number of rows      : 4.000.000
Batch size         : 0
Table Locking      : Activated
Transaction        : Disabled

Produce data       : 53.105 milliseconds
Consume data       : 96.660 milliseconds
Log file usage     : 4,18 Mb
```

The network graph is very similar to the previous one but the time needed to complete the operation is pretty higher than before. The reason is that – we guess – sorting takes time and the sort algorithm is sensible to the kind of data it receives.

One interesting point is that, even if the log file does not grow, the bulk insert operation on a clustered table takes a lot more time when compared to the heap. The reason is that sorting happens in the tempdb, where rows are temporarily stored before reaching the final database. In fact, during loading we will notice a tempdb growth of the same size of the data inserted into the table. The same loading process, when executed against a heap, does not make tempdb grow. This means that – loading a clustered table – we have to write the same data twice and this accounts for the longer time.

Up to now, we have inserted rows using a single producer and a single consumer. What will happen if we increase the number of producers and consumers and the destination table is a clustered table? We first tried with 4 producers and 4 consumers, here are the results:

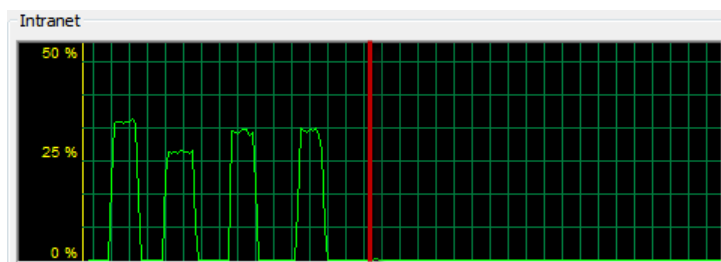
```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
```

```
Number of producers: 4
Number of consumers: 4
Number of rows      : 4.000.000
Batch size         : 0
Table Locking      : Activated
Transaction        : Disabled

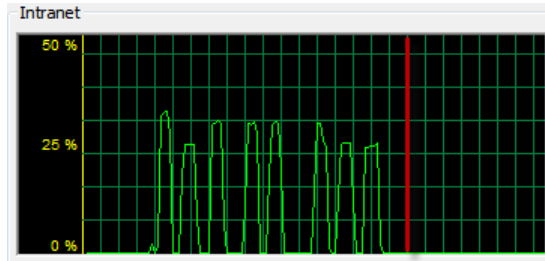
Produce data       : 35.020 milliseconds
Consume data       : 79.855 milliseconds
Log file usage     : 1.772,21 Mb
```

Please note that the log file usage is really huge. It seems that the whole loading process has been logged, even if the table was empty before loading. This is completely different, when compared to the single consumer test.

It is very interesting to observe the network usage graph too:



We see exactly four runs, each of which is very similar to the loading from a single producer: there is a peak when data is transferred and then a pause when data is sorted and inserted into the table. Looking at the tempdb, we notice that it does not grow as before. Something interesting is happening here... let us investigate a bit more. If we increase the number of consumers to 8, the time needed to load the table does not decrease and the graph seems to clearly show a pattern:



Now we have 8 peaks, one for each bulk copy thread. If we look at the activity monitor of the server during the bulk load operation, this is what we see:

Session ID	User Process	Command	Task State	Wait Type	Wait Resource	Blocked By	Wait Time (ms)
51	1	SELECT	RUNNING				
52	1						
53	1	BULK INSERT	RUNNING				
61	1						
54	1	BULK INSERT	SUSPENDED	LCK_M_X	objectlock lockPartition=0 ...		53
54	1	BULK INSERT	SUSPENDED	LCK_M_X	objectlock lockPartition=0 ...		53
54	1	BULK INSERT	SUSPENDED	LCK_M_X	objectlock lockPartition=0 ...		53
54	1	BULK INSERT	SUSPENDED	LCK_M_X	objectlock lockPartition=0 ...		53
54	1	BULK INSERT	SUSPENDED	LCK_M_X	objectlock lockPartition=0 ...		53

It is evident that the TABLOCK imposed by the first bulk insert operation effectively blocks all other bulk insert operations. Thus, if we increase the number of consumers on a clustered table, we do not gain anything from the performance point of view. Worse, the first bulk insert operation works on an empty clustered table, so it is minimally logged, but all the remaining ones will find the table already filled with data and so they will perform the loading in full logging mode, thus giving the bad performance we have noticed.

To double check this, we monitored the bulk insert on the heap:

Session ID	User Process	Command	Task State	Wait Type	Wait Resource	Blocked By	Wait Time (ms)
51	1	SELECT	RUNNING				
52	1						
53	1	BULK INSERT	RUNNABLE				
58	1	BULK INSERT	RUNNING	ASYNC_NETWORK_IO			
59	1						
61	1						
54	1	BULK INSERT	SUSPENDED	ASYNC_NETWORK_IO	External ExternalResource...		
55	1	BULK INSERT	RUNNABLE	ASYNC_NETWORK_IO	External ExternalResource...		
56	1	BULK INSERT	SUSPENDED	ASYNC_NETWORK_IO	External ExternalResource...		

We can see that, loading a heap, all processes will run in parallel, providing a much faster loading process. The loading of 4 million rows took 11.3 seconds.

So, if we load 4 million rows in a heap using 8 threads we can do that in 11 seconds. Then we can create the clustered index, and that takes 7 seconds, and we reach a final result of 18 seconds which, compared to the 80 seconds of loading into a table with the clustered index active, leads us to say that loading into a clustered table leads to such a poor performance that it is always better to first drop the index and re-

create it afterwards. The gain in performance is tremendous. Clearly, in order to create the clustered index we need enough space in the tempdb to sort the table but this is exactly the space needed for loading into the clustered table, so, at the end, we will not need more tempdb to load the heap and sort it later.

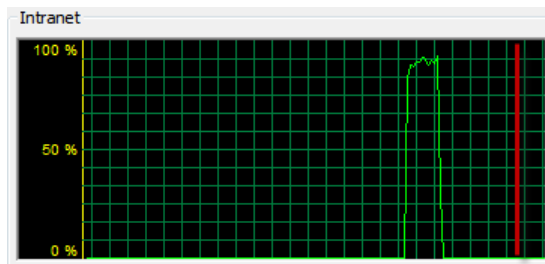
Nevertheless, since we have found that clustered loading leads to poor performance due to the TABLOCK, we decided to perform some others tests removing the TABLOCK option. The result is really interesting:

```
SqlBulkCopy tester - (C) 2009 SQLBI.COM
```

```
Number of producers: 8
Number of consumers: 8
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Disabled
Transaction         : Disabled

Produce data        : 37.766 milliseconds
Consume data        : 46.359 milliseconds
Log file usage      : 2.132,25 Mb
```

If we remove the TABLOCK option we get full logging but, since all processes will run in parallel, the overall time is reduced by a half. This is evident by looking at the network usage graph for this run:



We can see that all the data has been sent using the full network bandwidth with only one peak. The remaining time is needed by SQL Server to log, sort and write the data.

Trace Flag 610

It is now time to check if we should set trace flag 610 to on or off. This flag causes SQL Server to perform minimal logging when we load a clustered table which is not empty. The effect of this is that all rows that will be loaded into new pages will benefit from minimal logging while rows which will be inserted into existing pages will perform full logging.

The only situation where this flag might be useful, in our pattern, is when we perform a parallel bulk load into a clustered table and this table is not empty. Sadly to say, TF 610 works only if we impose a TABLOCK on the table and we have already seen that this causes the parallel bulk insert to be serialized by SQL Server, so we do not expect any big advantage by doing that.

Running the test we get this result:

```

SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 8
Number of consumers: 8
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Activated
Transaction         : Disabled

Produce data        : 3.781 milliseconds
Consume data        : 71.299 milliseconds
Log file usage      : 19,61 Mb

```

The final execution time is very similar to the parallel bulk insert on the clustered table. The big difference is in the log file usage, which is very small when compared with the previous value of 1.7Gb. Nevertheless, we have already seen that using TABLOCK and parallel loading on the clustered table gives us the worst performance in loading, so, for the specific pattern of bulk loading that we are testing, TF 610 is not really useful because we would prefer to remove the TABLOCK option and pay full logging in order to be able to parallel load data inside the table.

Bulk Insert and Indexes

It is useless to stress the fact that bulk loading a table which has active indexes is a bad practice. The time needed to perform updates to the index structure is huge and generates a lot of fragmentation. The best practice is to disable indexes before bulk load, add the data and then rebuild the indexes. Nevertheless, since we are evaluating performances in a pure speculative way, it is interesting to take a look at how the presence of indexes influence the loading process.

What we are guessing is that, in this specific situation, a change in the value of BatchSize might cause the loading process to perform better since the index restructuration is performed (and logged) in smaller chunks of data. Time to test now: we created two non clustered indexes on the heap:

```

CREATE NONCLUSTERED INDEX TestIndexInt ON Test (FieldInt1, FieldInt2, FieldInt3)
CREATE NONCLUSTERED INDEX TestIndexVarChar ON Test (FieldChar1, FieldChar2, FieldChar3)

```

The first index is pretty simple, based on three integers, the second one is very heavy (the key is 300 bytes long). We performed the loading of 4 million rows on an empty table and then loaded another four millions on the table already filled. All data is generated randomly so that each row is different, in order to stress the index restructuration process.

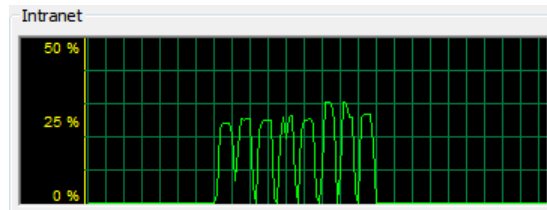
First of all, we loaded the table and measured the time needed to build the indexes:

Index	4 Million	8 Millions
Int	2 seconds	3 seconds
VarChar	13 seconds	26 seconds

Having done that, we want now to truncate the table and load data with the indexes active. We performed the tests with one index at a time active and then with both indexes activated. The base time (without indexes) is 12 seconds to load 4 million rows into the heap.

Index	First Load (4M)	Second Load (4M)	First Load + reindex	Second Load + reindex
Int	55 seconds	58 Seconds	12 + 2 = 14 seconds	12 + 3 = 15 seconds
VarChar	131 Seconds	215 Seconds	12 + 13 = 25 seconds	12 + 26 = 38 seconds
Int + VarChar	187 Seconds	298 Seconds	12 + 15 = 27 seconds	12 + 29 = 41 seconds

We see that the increase in time is really huge. Moreover, looking at the network usage graph, we see an old friend. This is the network graph of the first load (4 million with only int index active):



The bulk insert operations are serialized by SQL Server due to the sole presence of a non clustered index. This is confirmed looking at the activity monitor. Moreover, looking at the log file usage, we discover that it is increased enormously:

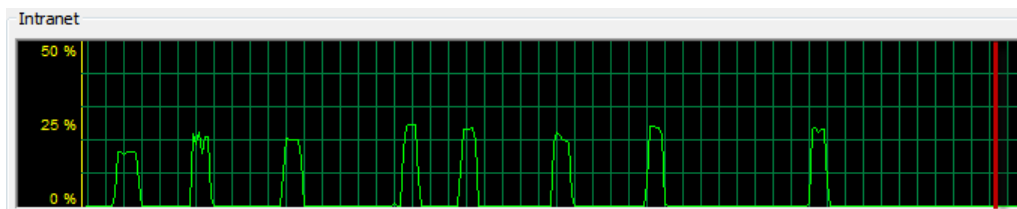
```

SqlBulkCopy tester - (C) 2009 SQLBI.COM
Number of producers: 8
Number of consumers: 8
Number of rows      : 4.000.000
Batch size          : 0
Table Locking       : Activated
Transaction         : Disabled

Produce data        : 37.597 milliseconds
Consume data        : 55.156 milliseconds
Log file usage      : 312,68 Mb

```

We already know that the only logged data is relative to the indexes but is really surprising to note that 312Mb have been logged against less than six megabytes when we had no indexes. The graphs and log sizes of all subsequent loads are worse and worse, as the process gets more complex. It is pretty easy to understand that rebuilding an index on a 300 byte key is much more complex than rebuilding an index base on three integers. This is evident by looking at the network graph for the second bulk insert with both indexes active, where SQL Server had to update two complex indexes after each bulk insert and thus introduces a huge time, after each peak, to restructure the indexes:



Having seen that, and knowing that we can increase parallelism by removing the TABLOCK option, we tested again without the TABLOCK option:

Index	First Load (4M)	Second Load (4M)
Int	55 seconds	60 Seconds
VarChar	145 Seconds	146 Seconds
Int + VarChar	196 Seconds	182 Seconds

Removing the table locking all the thread run in parallel, showing some locks during the final phase of index restructuration. The log file usage is increased a lot and, surprisingly, the overall time has not changed, showing that – if there are indexes – removing table locking seems not very useful. The only load that gained some time is the one where both indexes are active.

The last test is to check whether different values for the BatchSize parameter will influence in some way the loading. We used no table locking and measured only the first load, as we have seen that there are no big differences between the first and the second load if table locking is disable.

The results are really surprising:

Batch Size	100	1.000	2.000	5.000
Int	73	38	37	62
VarChar	211	53	73	49
Int + VarChar	273	98	56	91

A value of batch size between 1.000 and 2.000 produces nice results, great when compared with the same results obtained with batch size left to zero. This is in contrast with what we have measured with the heap with no indexes, where a batch size set to zero got the best performance.

This test was just a speculative one. The conclusion is that, when we need to perform the bulk loading of data, indexes are a big problem and cannot be left active. But, if for any reason we do need to leave the indexes active, then running all the processes in parallel, with no table locking and a defined value for the batch size will give us pretty good performance. Clearly, the best value for the batch size parameter should be determined after some trials since, as we can see from the table, it depends on how many indexes are present.

Final considerations

After all these tests, it is now time to draw some conclusions:

- Heap: the best way to load a heap is to provide parallel SqlBulkCopy operations. If the heap has no indexes then we can simply load data inside it. If there are indexes active, then it is normally better to drop them and recreate them at the end.
- Clustered Table: if it is feasible, the best way to load it is to remove the clustered index, load data inside it as a heap and then rebuild the clustered index afterwards. If this is not feasible, then loading data without TABLOCK in parallel threads leads to good performances even if they are much worse when compared with the heap method.

- Indexes: as we have seen, indexes create huge problems with the parallelism so is it always a good idea to load without any indexes active. If this is not feasible, then adjusting the Batch Size parameter will lead to pretty good performances. This is the only situation where we encountered a good impact from the settings of Batch Size.

We did not discuss partitioning, which might be used with great success since, if we are able to load data inside a partition of our final destination table, then we can create it as a heap, load data in the fastest way, then rebuild all indexes (clustered and non clustered) and switch the partition in, reaching the best performance in data loading.