The Effect of NOLOCK on Performance

Introduction - How locking works

Using the NOLOCK query optimiser hint is generally considered good practice in order to improve concurrency on a busy system. When the NOLOCK hint is included in a SELECT statement, no locks are taken when data is read. The result is a Dirty Read, which means that another process could be updating the data at the exact time you are reading it. There are no guarantees that your query will retrieve the most recent data.

The advantage to performance is that your reading of data will not block updates from taking place, and updates will not block your reading of data. This means that you will have improved concurrency on your system, and more tasks can be performed at the same time. Without NOLOCK each access to the database results in a lock being made on the row being read, or the page on which the data is located. SELECT statements take Shared (Read) locks. This means that multiple SELECT statements are allowed simultaneous access, but other processes are blocked from modifying the data. The updates will queue until all the reads have completed, and reads requested after the update will wait for the updates to complete. The result to your system is delay, otherwise known as Blocking.

Blocks often take milliseconds to resolve themselves, and this is not always noticeable to the end-user. At other times, users are forced to wait for the system to respond. Performance problems can be difficult to diagnose and resolve, but blocking is often easily resolved by the use of NOLOCK.

One solution to blocking is to place the NOLOCK query optimiser hint on all select statements where a dirty read is not going to cause problems for your data integrity. You would generally avoid this practice in your Finance stored procedures, for example.

Faster reading of data with NOLOCK

Logically, a select using NOLOCK should return data faster than without, as the taking of locks invokes an overhead. Ever curious, I decided to investigate the impact of NOLOCK on the performance of a SELECT query. In order to prove this I needed data - lots of it. The following example shows code I ran to create two tables: Products and Orders, and populate them with 5 million rows each.

You will need to create a database called NOLOCKPerformanceTest, and I recommend you allocate 400MB of initial space for the data file. This code could take over an hour to complete, depending on your system. I am running this on an Intel Dual Core 2Ghz laptop, with a 7200rpm SATA drive and 2GB RAM. The database edition is SQL Server 2005 Developer, with SP1 and hotfix 2153 installed.

I decided to use very large tables, as the longer it takes my test to run, the more noticeable any differences.

```
USE NOLOCKPerformanceTest
GO
```

```
CREATE TABLE Orders (
                OrderID         int IDENTITY(1,1) PRIMARY KEY,
                ProductID       int NOT NULL,
                OrderDate       varchar(50) NOT NULL
                )


DECLARE @i                      int,
        @OrderDate              datetime,
        @DateDifferenceDays     int,
        @ProductID              int

SET @i = 1
SET @DateDifferenceDays = 0
SET @ProductID = 0

WHILE @i < 5000000
BEGIN

        -- The order date will decrease by one day every 1000 orders
        -- Drop the time using CONVERT
        SET @OrderDate = CONVERT(datetime, CONVERT(varchar (10), getdate() - @DateDi
        IF @i % 1000 = 0
                SET @DateDifferenceDays = @DateDifferenceDays + 1

        -- Ensure the Referential Integrity with the Product tables
        SET @ProductID = @ProductID + @i
        IF @ProductID > 500000
                SET @ProductID = 1

        INSERT  Orders (ProductID, OrderDate)
        VALUES  (@ProductID , @OrderDate)

        SET @i = @i + 1

END

GO
```

Example 1 Load table Orders

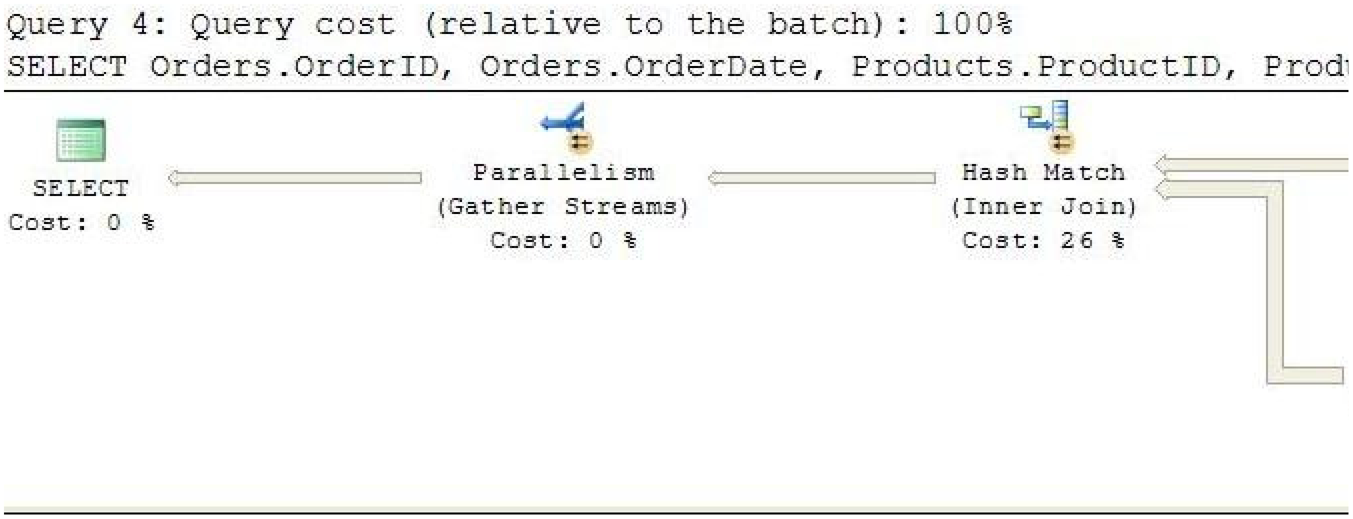
```
USE NOLOCKPerformanceTest
GO

CREATE TABLE Products (
                ProductID       int IDENTITY(1,1) PRIMARY KEY,
                ProductDescription varchar(100) NOT NULL,
                ProductPrice    money
                )



DECLARE @i                      int,
        @ProductDescription     varchar(100)

SET @i = 1

WHILE @i < 5000000
```

```
BEGIN

        SET @ProductDescription = 'Product ' + CONVERT(varchar, @i)

        INSERT  Products (ProductDescription, ProductPrice)
        VALUES  (@ProductDescription , @i)

        SET @i = @i + 1

END

GO
```

Example 2 Load table Products

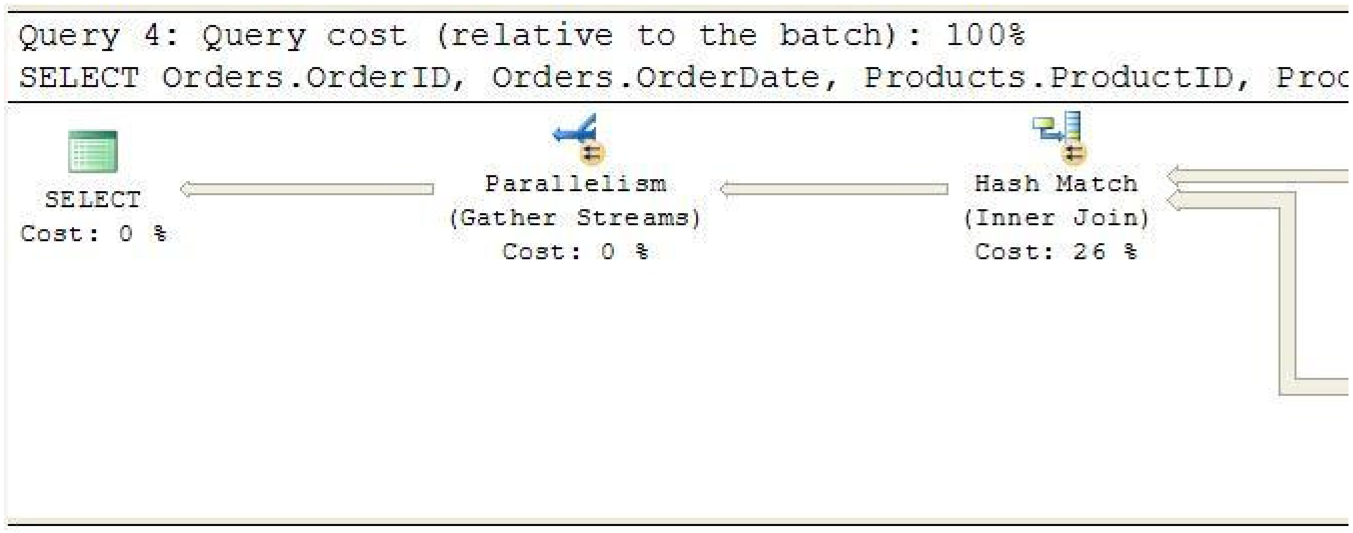
I started the exercise by running a simple join query between the two tables, while checking the runtime with NOLOCK, and without NOLOCK. I flush the data buffer before each execution, and I drew an average over 5 executions.

```
USE NOLOCKPerformanceTest
GO

DBCC DROPCLEANBUFFERS

DECLARE @StartTime datetime

SET @StartTime = getdate()

SELECT  Orders.OrderID,
        Orders.OrderDate,
        Products.ProductID,
        Products.ProductDescription
FROM    Orders
        INNER JOIN Products
        ON orders.ProductID = Products.ProductID
WHERE   Orders.OrderDate BETWEEN CONVERT(datetime, '01 Jan 2006') and getdate()

SELECT getdate() - @StartTime
GO
```


```
Query 4: Query cost (relative to the batch): 100%
SELECT Orders.OrderID, Orders.OrderDate, Products.ProductID, Prod
```

Example 3 Runtime test without NOLOCK with the estimated execution plan

```
USE NOLOCKPerformanceTest
GO

DBCC DROPCLEANBUFFERS

DECLARE @StartTime datetime

SET @StartTime = getdate()

SELECT  Orders.OrderID,
        Orders.OrderDate,
        Products.ProductID,
        Products.ProductDescription
FROM    Orders WITH (NOLOCK)
        INNER JOIN Products WITH (NOLOCK)
        ON orders.ProductID = Products.ProductID
WHERE   Orders.OrderDate BETWEEN CONVERT(datetime, '01 Jan 2006') and getdate()

SELECT getdate() - @StartTime
GO
```



```
Query 4: Query cost (relative to the batch): 100%
SELECT Orders.OrderID, Orders.OrderDate, Products.ProductID, Proc
```

Example 4 Runtime test with NOLOCK with the estimated execution plan

The Results

The results for execution times are listed below. It takes a while to run, as 310,001 rows are returned. If you are running these queries, execution times will vary depending on the specification of your server. The format for these results is hh:mm:ss.nnn, where nnn is milliseconds.

Without NOLOCK:

Run1: 00:00:29.470

Run2: 00:00:30.467

Run3: 00:00:28.877

Run4: 00:00:29.123

Run5: 00:00:29.407

Average: 00:00:29:469

With NOLOCK:

Run1: 00:00:25.060

Run2: 00:00:25.157

Run3: 00:00:25.107

Run4: 00:00:25.140

Run5: 00:00:25.893

Average: 00:00:25:271

This test shows the average execution time is less when using NOLOCK. The average time saved is 00:00:04.197, which equates to a saving in elapsed time of approximately 14%.

My next step was to check the amount of CPU and IO used in both queries. I added the following code immediately before the first SELECT in examples 3 and 4, and ran both queries again:

```
SET STATISTICS IO ON
SET STATISTICS TIME ON
```

The execution times were the same as before, and the results for the SET STATISTICS options were as follows:

Without NOLOCK:

Table 'Orders'. Scan count 3, logical reads 27266, physical reads 10, read-ahead reads 24786, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Products'. Scan count 3, logical reads 27119, physical reads 78, read-ahead reads 24145, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 13844 ms, elapsed time = 27480 ms.

With NOLOCK:

```
Table 'Orders'. Scan count 3, logical reads 24845, physical reads 0, read-ahead reads 24826, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Products'. Scan count 3, logical reads 24711, physical reads 0, read-ahead reads 24689, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 13813 ms, elapsed time = 24999 ms.
```

While not being an expert at IO, I nevertheless noticed lower physical and logical reads for the NOLOCK version. I ran both queries several times, and noticed the same pattern. Using NOLOCK definitely results in lower IO.

The CPU time is similar between the versions, though you can see that once again the elapsed time is lower when using NOLOCK.

Changing the date in the WHERE clause (see examples 3 and 4) to "01 Dec 2006" results in no rows being returned. However, the execution time for the without NOLOCK example shows an average of about 24 seconds. With NOLOCK it runs for approximately 22 seconds each time the query is run.

Conclusion

This article has hopefully proven that using NOLOCK when querying data results in lower IO and faster response times. I have executed these queries at least a dozen times each over several weeks, and the expected result is always the same.

If you can suggest any additional queries I can put to the test, please don't hesitate to contact me on wfillis@gmail.com. I would be happy to test your scenarios, and present the results in a future article.