**[SELECT Hints, Tips, Tricks FROM Hugo Kornelis WHERE RDBMS = 'SQL Server'](http://sqlblog.com/blogs/hugo_kornelis/default.aspx)**

<http://sqlblog.com/blogs/hugo_kornelis/archive/2007/11/21/curious-cursor-optimization-options.aspx>

Based on all tests, it turns out that the **best performance is achieved by specifying a STATIC cursor.**

**I would add the LOCAL, FORWARD\_ONLY, and READ\_ONLY options for documentation purposes, but they make no performance difference.**

With these options, execution time went down from 6.3 to 9 seconds (depending on the ORDER BY) to 3.3 to 3.4 seconds. Of course, none of those come even close to the 0.2 seconds of the set-based equivalent for this test case:

**Curious cursor optimization options**

The best way to optimize performance of a cursor is, of course, to rip it out and replace it with set-based logic. But there is still a small category of problems where a cursor will outperform a set-based solution. The introduction of ranking functions in SQL Server 2005 has taken a large chunk out of that category – but some remain. For those problems, it makes sense to investigate the performance effects of the various cursor options.

I am currently preparing a series of blog posts on a neat set-based solution I found for a problem that screams “cursor” from all corners. But in order to level the playing field, I figured that it would be only fair to optimize the hell out of the cursor-based solution before blasting it to pieces with my set-based version. So I suddenly found myself doing something I never expected to do: finding the set of cursor options that yields the best performance.

That task turned out to be rather time-consuming, as there are a lot of cursor options that can all be combined in a huge number of ways. And I had to test all those combinations in various scenarios, like reading data in a variety of orders, and updating data in two separate ways. I won’t bore you with all the numbers here; instead, I intend to point out some highlights, including some very curious finds. For your reference, I have included a spreadsheet with the results of all test as an attachment to this post.

Disclaimer: All results presented here are only valid for my test cases (as presented below) on my test data (a copy of the SalesOrderDetail table in the AdventureWorks sample database), on my machine (a desktop with 2GB of memory, a dual-core processor, running SQL Server 2005 SP2), and with my workload (just myself, and only the test scripts were active). If your situation is different, for instance if the table will not fit in cache, if the database is heavily accessed by competing processes, or if virtually any other variable changes, you really ought to perform your own test if you want to squeeze everything out of your cursor. And also consider that many options are included to achieve other goals than performance, so you may not be able to use all options without breaking something.

**Reading data**

Many cursors are used to create reports. The data read is ordered in the order required for the report, and running totals and subtotals are kept and reset as required while reading rows. Those already on SQL Server 2005 can often leverage the new ranking functions to calculate the same running totals without the overhead of a cursor, but if you are still stuck on SQL Server 2000 or if you face a problem that the ranking functions can’t solve, you may find yourself preferring a cursor over the exponentially degrading performance of the correlated subquery that the set-based alternative requires.

Since the order of these cursors is dictated by the report requirements rather than the table and index layout, I decided to test the three variations you might encounter – you may be so lucky that the order of the report matches the clustered index, or you might find that a nonclustered index matches the order you need, or you may be so unlucky that you need to order by a column that is not indexed.

I used the code below for my performance tests. You can run this code as is on the AdventureWorks sample database, or you can do as I did and copy the Sales.SalesOrderDetail table, with all indexes and all data, to your own testing database.

-- Keep track of execution time

DECLARE @start datetime;

SET @start = CURRENT\_TIMESTAMP;

-- Declare and initialize variables for cursor loop

DECLARE @SalesOrderID int,

@SalesOrderDetailID int,

@OrderQty smallint,

@ProductID int,

@LineTotal numeric(38,6),

@SubTotal numeric(38,6);

SET @SubTotal = 0;

-- Declare and init cursor

DECLARE SalesOrderDetailCursor

CURSOR

LOCAL -- LOCAL or GLOBAL

FORWARD\_ONLY -- FORWARD\_ONLY or SCROLL

STATIC -- STATIC, KEYSET, DYNAMIC, or FAST\_FORWARD

READ\_ONLY -- READ\_ONLY, SCROLL\_LOCKS, or OPTIMISTIC

TYPE\_WARNING -- Inform me of implicit conversions

FOR SELECT SalesOrderID, SalesOrderDetailID,

OrderQty, ProductID, LineTotal

FROM Sales.SalesOrderDetail

ORDER BY SalesOrderID, SalesOrderDetailID; -- Match clustered index

-- ORDER BY ProductID; -- Match nonclustered index

-- ORDER BY LineTotal; -- Doesn’t match an index

OPEN SalesOrderDetailCursor;

-- Fetch first row to start loop

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

-- Process all rows

WHILE @@FETCH\_STATUS = 0

BEGIN;

-- Accumulate total

SET @SubTotal = @SubTotal + @LineTotal;

-- Fetch next row

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

END;

-- Done processing; close and deallocate to free up resources

CLOSE SalesOrderDetailCursor;

DEALLOCATE SalesOrderDetailCursor;

-- Display result and duration

SELECT @SubTotal;

SELECT DATEDIFF(ms, @start, CURRENT\_TIMESTAMP);

go

The first surprise came straight when I set my baseline by commenting out all options of the DECLARE CURSOR statement. The execution time when ordering by the clustered index was 6.9 seconds; when ordering by a nonclustered index it was 9 seconds – but when ordering by an unindexed column, the cursor with default options turned out to be *faster*, at only 6.4 seconds. I later found the reason for this to be that the first two defaulted to a relatively slow dynamic cursor, whereas the latter used the faster technique of a keyset cursor.

Choosing LOCAL or GLOBAL had no effect on cursor performance. This was as expected, since this option only controls the scope of the cursor, nothing else. For this reason, I excluded this option from testing the variants for updating with a cursor.

I didn’t see any difference between the FORWARD\_ONLY and SCROLL options either. This came as a surprise, since FORWARD\_ONLY exposes only a subset of the functionality of the SCROLL version. I really expected SQL Server to be able to do some clever optimization if it knew that I’d never read in any other direction than from the first to the last row. I’m really wondering why the FORWARD\_ONLY option is not deprecated, seeing that there is no advantage at all in specifying it – but maybe the development team in Redmond knows something I don’t?

The static, keyset, and dynamic cursors performed exactly as expected – in all cases, the static cursor was the fastest, the keyset came second, and the dynamic cursor finished last. No surprises here – until I started my tests with the cursor that orders by an unindexed column. In these tests, SQL Server informed be (due to the TYPE\_WARNING option) that the created cursor was not of the requested type. It did not tell me what type it did create, nor why it disregarded the requested options. I failed to see anything in Books Online to explain this behavior, so [I filed a bug for this](https://connect.microsoft.com/SQLServer/feedback/ViewFeedback.aspx?FeedbackID=311452). This did explain why the “hardest” sort option was the fastest when running with default options – since a dynamic cursor was not available, this one had to use a keyset cursor instead.

My biggest surprise came when I tested the FAST\_FORWARD option. According to Books Online, this option “specifies a FORWARD\_ONLY, READ\_ONLY cursor with performance optimizations enabled”, so I expected performance to be at least on par with, and probably better than that of a STATIC FORWARD\_ONLY READ\_ONLY cursor – but instead, the FAST\_FORWARD option turned out to be consistently slower, in some cases even by 15%!

The last set of options, the ones specifying the locking behavior, turned out to depend on the chosen cursor type. For a static cursor, the two available options made no difference. For other cursors, READ\_ONLY was best – but SCROLL\_LOCKS was second for keyset cursors and third for dynamic cursors, and OPTIMISTIC was second for dynamic and third for keyset. Go figure.

Based on all tests, it turns out that the **best performance is achieved by specifying a STATIC cursor.** **I would add the LOCAL, FORWARD\_ONLY, and READ\_ONLY options for documentation purposes, but they make no performance difference.** With these options, execution time went down from 6.3 to 9 seconds (depending on the ORDER BY) to 3.3 to 3.4 seconds. Of course, none of those come even close to the 0.2 seconds of the set-based equivalent for this test case:

-- Keep track of execution time

DECLARE @start datetime;

SET @start = CURRENT\_TIMESTAMP;

-- Calculate and display result

SELECT SUM(LineTotal)

FROM Sales.SalesOrderDetail;

-- Display duration

SELECT DATEDIFF(ms, @start, CURRENT\_TIMESTAMP);

go

**Modifying data**

Another scenario in which cursors are used is when data has to be updated, and the calculation to determine the new data is thought to be to complicated for a set-based approach. In those cases, a cursor is used to process the rows one by one, calculate the new data, and update the data with the calculation results.

If you specify the FOR UPDATE clause in the cursor declaration, you can use the WHERE CURRENT OF clause of the UPDATE command to update the last row fetched. Of course, you can also omit the FOR UPDATE clause and use a regular UPDATE statement, using the primary key values of the row just read to find the row to update.

Since I expected a FOR UPDATE cursor to be optimized for updating the last row fetched, I first tested its performance, by using this code:

-- Enclose in transaction so we can roll back changes for the next test

BEGIN TRANSACTION;

go

-- Keep track of execution time

DECLARE @start datetime;

SET @start = CURRENT\_TIMESTAMP;

-- Declare and initialize variables for cursor loop

DECLARE @SalesOrderID int,

@SalesOrderDetailID int,

@OrderQty smallint,

@ProductID int,

@LineTotal numeric(38,6);

-- Declare and init cursor

DECLARE SalesOrderDetailCursor

CURSOR

LOCAL -- LOCAL or GLOBAL makes no difference for performance

FORWARD\_ONLY -- FORWARD\_ONLY or SCROLL

KEYSET -- KEYSET or DYNAMIC

-- (other options are incompatible with FOR UPDATE)

SCROLL\_LOCKS -- SCROLL\_LOCKS or OPTIMISTIC

-- (READ\_ONLY is incompatible with FOR UPDATE)

TYPE\_WARNING -- Inform me of implicit conversions

FOR SELECT SalesOrderID, SalesOrderDetailID,

OrderQty, ProductID, LineTotal

FROM Sales.SalesOrderDetail

ORDER BY SalesOrderID, SalesOrderDetailID

FOR UPDATE -- FOR UPDATE or FOR UPDATE OF OrderQty

;

OPEN SalesOrderDetailCursor;

-- Fetch first row to start loop

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

-- Process all rows

WHILE @@FETCH\_STATUS = 0

BEGIN;

-- Change OrderQty of current order

UPDATE Sales.SalesOrderDetail

SET OrderQty = @OrderQty + 1

WHERE CURRENT OF SalesOrderDetailCursor;

-- Fetch next row

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

END;

-- Done processing; close and deallocate to free up resources

CLOSE SalesOrderDetailCursor;

DEALLOCATE SalesOrderDetailCursor;

-- Display duration

SELECT DATEDIFF(ms, @start, CURRENT\_TIMESTAMP);

go

-- Rollback changes for the next test

ROLLBACK TRANSACTION;

go

Just as with the tests that only read the data, there was no difference between SCROLL and FORWARD\_ONLY cursors. And just as with the tests that only read the data, KEYSET cursors were consistently faster than their DYNAMIC counterparts. However, in this case the SCROLL\_LOCKS locking option turned out to be consistently faster than OPTIMISTIC, though I expect that this might change if only a fraction of the rows is updated.

From a performance point of view, there is absolutely no difference between a generic FOR UPDATE or a completely specified FOR UPDATE OF *column*, *column*, … For documentation purposes, I would prefer the latter.

And again, just as with the tests that only read the data, the default cursor options chosen when I did not specify any turned out to select the slowest of all available options. Ugh!

However, the real kicker came when I left out the FOR UPDATE clause of the CREATE CURSOR statement and changed the UPDATE statement to use the primary key values instead of the WHERE CURRENT OF clause. One would expect that this clause would be fast – since it is written especially for, and can be used exclusively in, the processing of a FOR UPDATE cursor, every trick in the book can be used to optimize this. However, the reverse turned out to be true. Even the fastest of all WHERE CURRENT OF variations I tested was easily beaten by even the slowest of all WHERE PrimaryKey = @PrimaryKey variations. Here is the code I used, in case you want to test it yourself:

-- Enclose in transaction so we can roll back changes for the next test

BEGIN TRANSACTION;

go

-- Keep track of execution time

DECLARE @start datetime;

SET @start = CURRENT\_TIMESTAMP;

-- Declare and initialize variables for cursor loop

DECLARE @SalesOrderID int,

@SalesOrderDetailID int,

@OrderQty smallint,

@ProductID int,

@LineTotal numeric(38,6);

-- Declare and init cursor

DECLARE SalesOrderDetailCursor

CURSOR

LOCAL -- LOCAL or GLOBAL makes no difference for performance

FORWARD\_ONLY -- FORWARD\_ONLY or SCROLL

STATIC -- STATIC, KEYSET, DYNAMIC, or FAST\_FORWARD

READ\_ONLY -- READ\_ONLY, SCROLL\_LOCKS, or OPTIMISTIC

TYPE\_WARNING -- Inform me of implicit conversions

FOR SELECT SalesOrderID, SalesOrderDetailID,

OrderQty, ProductID, LineTotal

FROM Sales.SalesOrderDetail

ORDER BY SalesOrderID, SalesOrderDetailID;

OPEN SalesOrderDetailCursor;

-- Fetch first row to start loop

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

-- Process all rows

WHILE @@FETCH\_STATUS = 0

BEGIN;

-- Change OrderQty of current order

UPDATE Sales.SalesOrderDetail

SET OrderQty = @OrderQty + 1

WHERE SalesOrderID = @SalesOrderID

AND SalesOrderDetailID = @SalesOrderDetailID;

-- Fetch next row

FETCH NEXT FROM SalesOrderDetailCursor

INTO @SalesOrderID, @SalesOrderDetailID,

@OrderQty, @ProductID, @LineTotal;

END;

-- Done processing; close and deallocate to free up resources

CLOSE SalesOrderDetailCursor;

DEALLOCATE SalesOrderDetailCursor;

-- Display duration

SELECT DATEDIFF(ms, @start, CURRENT\_TIMESTAMP);

go

-- Rollback changes for the next test

ROLLBACK TRANSACTION;

go

So from using WHERE CURRENT OF and default options, at 16.6 seconds, I’ve gotten execution time down to 5.1 seconds by using the primary key for the update and specifying a STATIC cursor (including the LOCAL, FAST\_FORWARD, and READ\_ONLY options for documentation). Looks good, as long as I close my eyes to the 0.4 second execution time of the set-based version:

-- Enclose in transaction so we can roll back changes for the next test

BEGIN TRANSACTION;

go

-- Keep track of execution time

DECLARE @start datetime;

SET @start = CURRENT\_TIMESTAMP;

-- Change OrderQty of all orders

UPDATE Sales.SalesOrderDetail

SET OrderQty = OrderQty + 1;

-- Display duration

SELECT DATEDIFF(ms, @start, CURRENT\_TIMESTAMP);

go

-- Rollback changes for the next test

ROLLBACK TRANSACTION;

go

**Conclusion**

If you have to optimize a cursor for performance, keep the following considerations in mind:

1. Always try to replace the cursor by a set-based equivalent first. If you fail to see how, do not hesitate to ask in one of the SQL Server newsgroups.
2. If you are really stuck with a cursor, then do NOT rely on the default options. They will result in the slowest of all possible option combinations
3. If you think that the FAST\_FORWARD option results in the fastest possible performance, think again. I have not found one single test case where it was faster than, or even as fast as, a STATIC cursor.
4. Do NOT use the WHERE CURRENT OF syntax of the UPDATE command. Using a regular WHERE clause with the primary key values will speed up your performance by a factor of two to three.
5. Do not rely blindly on my performance results. Remember, the one thing that is always true when working with SQL Server is: “it depends”.

Published Wednesday, November 21, 2007 2:15 AM by [Hugo Kornelis](http://sqlblog.com/user/Profile.aspx?UserID=13570)

Filed under: [Surprise](http://sqlblog.com/blogs/hugo_kornelis/archive/tags/Surprise/default.aspx), [Performance](http://sqlblog.com/blogs/hugo_kornelis/archive/tags/Performance/default.aspx), [Cursor vs Set](http://sqlblog.com/blogs/hugo_kornelis/archive/tags/Cursor+vs+Set/default.aspx)

**SQL Cursors - how to avoid them**

**http://www.sqlbook.com/SQL/Avoiding-using-SQL-Cursors-20.aspx**

**Introduction**

There may be times when you need to loop through a resultset a row at a time and perform a certain action for each row. The most obvious way to solve this task is to use a **SQL Cursor**.

Whilst cursors may seem like a good idea they can often cause your database application problems as they can **lock** the tables that are used to populate the cursor whilst the rows in the cursor are looped through. Depending on the action that you are performing on each row this can take a considerable time. The effect of this is that tables cannot be updated or accessed by other users whilst the cursor is open.

I know one SQL specialist who when interviewing for DBA roles asks the candidate to write the syntax for using a cursor. If the candidate knows it then it is a negative point against them as they shouldn't use cursors frequently enough so that they can remember the syntax.

Whilst many SQL books advise you not to use SQL cursors, not many provide alternative solutions. This article shows how alternatives to cursors can be implemented.

**An example of a SQL Cursor that we want to avoid**

DECLARE @CustomerID int

DECLARE @FirstName varchar(30), @LastName varchar(30)

-- declare cursor called ActiveCustomers

DECLARE ActiveCustomers Cursor FOR

SELECT CustomerID, FirstName, LastName

FROM Customer

WHERE Active = 1

-- Open the cursor

OPEN ActiveCustomers

-- Fetch the first row of the cursor and assign its values into variables

FETCH NEXT FROM ActiveCustomers INTO @CustomerID, @FirstName, @LastName

-- perform action whilst a row was found

WHILE @@FETCH\_STATUS = 0

BEGIN

Exec MyStoredProc @CustomerID, @Forename, @Surname

-- get next row of cursor

FETCH NEXT FROM ActiveCustomers INTO @CustomerID, @FirstName, @LastName

END

-- Close the cursor to release locks

CLOSE ActiveCustomers

-- Free memory used by cursor

DEALLOCATE ActiveCustomers

**Cursor alternative 1: Using the SQL WHILE loop**

SQL provides us with the WHILE looping structure. This can be utilised with a temporary table that enables us to avoid using a cursor:

-- Create a temporary table, note the IDENTITY

-- column that will be used to loop through

-- the rows of this table

CREATE TABLE #ActiveCustomer (

RowID int IDENTITY(1, 1),

CustomerID int,

FirstName varchar(30),

LastName varchar(30)

)

DECLARE @NumberRecords int, @RowCount int

DECLARE @CustomerID int, @FirstName varchar(30), @LastName varchar(30)

-- Insert the resultset we want to loop through

-- into the temporary table

INSERT INTO #ActiveCustomer (CustomerID, FirstName, LastName)

SELECT CustomerID, FirstName, LastName

FROM Customer

WHERE Active = 1

-- Get the number of records in the temporary table

SET @NumberRecords = @@ROWCOUNT

SET @RowCount = 1

-- loop through all records in the temporary table

-- using the WHILE loop construct

WHILE @RowCount <= @NumberRecords

BEGIN

SELECT @CustomerID = CustomerID, @FirstName = FirstName, @LastName = LastName

FROM #ActiveCustomer

WHERE RowID = @RowCount

EXEC MyStoredProc @CustomerID, @FirstName, @LastName

SET @RowCount = @RowCount + 1

END

-- drop the temporary table

DROP TABLE #ActiveCustomer

We can see the above code gives the same functionality as the first code example but without using a cursor. This gives us the benefits that the Customer table is not locked as we are looping through our resultset so other queries on the Customer table that are submitted by other users will execute much faster. We will also have a faster operating SQL script by avoiding cursors which are slow in themselves.

**Cursor Alternative 2: Using User Defined Functions**

Cursors are sometimes used to perform a calculation on values that come from each row in its rowset. This scenario can also be achieved by replacing a Cursor with a User Defined Function. An example of a User Defined Function performing a calculation is given below:

-- return a discount %age that the customer

-- can recieve based on their no. and value

-- of purchases

CREATE FUNCTION dbo.GetDiscountLevel(

@CustomerID int

)

RETURNS int

AS

BEGIN

DECLARE @DiscountPercent int

DECLARE @NumberOrders int, @SalesTotal float

SELECT @NumberOrders = COUNT(OrderID),

@SalesTotal = SUM(TotalCost)

FROM Sales

WHERE CustomerID = @CustomerID

IF @SalesTotal > 5000.00 AND @NumberOrders > 5

SET @DiscountPercent = 5

ELSE

BEGIN

IF @SalesTotal > 3000.00 AND @NumberOrders > 3

SET @DiscountPercent = 3

ELSE

SET @DiscountPercent = 0

END

Return @DiscountPercent

END

An example of this function being used to replace a cursor might look something like:

SELECT FirstName, LastName, dbo.GetDiscountLevel(CustomerID) As DiscountPercent

FROM Customer

**Conclusion**

In this article we have seen how SQL Cursors can cause performance problems and affect other queries by locking tables. We have demonstrated two popular ways of avoiding the use of cursors. The WHILE loop does avoid the use of a Cursor but it still uses an iterative loop. The User Defined Function option keeps our SELECT query tidy and enables us to perform calculations using column values from our SELECT statement.

When attempting to redesign your code to avoid cursors you should always check the execution time of your scripts. Very occassionally you may find a cursor gives better performance than an alternative method of performing the same task. When doing this remember that the script execution time is not the only thing to check, impact on queries being run by other users at the same time is also a key factor.