Performance Implications of Various Cursor Types in Microsoft SQL Server

By: Edward Whalen
Performance Tuning Corporation

INTRODUCTION

There are a number of different types of cursors that can be created and used in Microsoft SQL Server 2000 and SQL Server 7.0. By choosing the most efficient cursor type for your requirements you can not only speed up the application, but you can conserve valuable system resources as well. These resources consist of CPU time, I/O capacity, memory utilization and bus bandwidth. Different cursor types are design for different needs. In this paper, the various cursor types will be discussed and their performance implications will be detailed.

What is a Cursor?

Typically an operation such as a SELECT statement returns a number of rows. Applications that perform operations on each row in the data set individually cannot easily take a rowset and operate effectively within the context of SQL statements. This rowset will have to be broken down within the application so that operations can take place on the individual rows.

A cursor allows you to individually retrieve the contents of a single row within a rowset within the context of SQL statements. A cursor is created using a SQL statement and the rows are returned to the application by using a fetch statement to retrieve the individual rows. This allows easy access to these individual rows, without having to create an array within your application and loading data from the SQL statement into the array.

There are various features of cursors that can be very useful. In addition to stepping through a cursor in one direction, it is also possible to move both directions within a cursor. In addition there are also features that allow easy access to modifications of the current row within the cursor. These features will be described later in this paper.

Uses of Cursors

Cursors are often used for operating on job queues or within large batch operations, however, sometimes innovative uses of cursors can also be used to improve performance. Depending on the type of operation, and the final outcome of the application you may or may not be able to use a cursor.

Several years ago I had the opportunity to greatly improve the performance of an application by changing a SELECT statement that used a TOP 1 command to a cursor based operation. The client had created their own queuing mechanism within their application and were using a SELECT statement with approximately 10 different predicates in the WHERE clause to find the one item on top of the queue. This operation was taking several seconds to complete and large resources on the system to perform this SELECT operation. Because of the columns used in the WHERE clause an index could not effectively be used.

Upon talking with the client, there were very few items that did not meet this criteria. So, I create a cursor that operated on a SELECT statement that only referenced a few columns in the WHERE clause and could effectively use the indexes that existed on that column. Once data was retrieved into the cursor other SQL statements were used to determine if the current row met the other criteria. If not, the next row was fetched. In this manner, an average of 4 fetches were used to retrieve the required row. At that point the cursor was closed.

By using a cursor and the proper indexes, the entire operation was reduced from several seconds to about 35 ms to complete the operation. This took the entire queuing operation from about 10 hours to less than 2 hours, thus allowing the nightly batch operations to complete in the required time.

Cursor Example

A cursor is created with the DECLARE CURSOR statement and opened with the OPEN CURSOR statement. The rows are read into the stored procedure or application using the FETCH command. Let's look at an example of a simple set of SQL statements that use a cursor.

```
DECLARE @LastName nvarchar(20),
        @FirstName nvarchar(10)

DECLARE EmpCursor CURSOR FOR
SELECT LastName, FirstName
FROM Northwind.dbo.Employees
ORDER BY LastName

OPEN EmpCursor

FETCH NEXT FROM EmpCursor INTO @LastName, @FirstName
        SELECT @LastName, @FirstName, EmployeeID, Extension
                FROM Northwind.dbo.Employees
                WHERE LastName = @LastName AND FirstName = @Firstname

WHILE @@FETCH_STATUS = 0
BEGIN
        FETCH NEXT FROM EmpCursor INTO @LastName, @FirstName
                SELECT @LastName, @FirstName, EmployeeID, Extension
                        FROM Northwind.dbo.Employees
                        WHERE LastName = @LastName AND FirstName = @Firstname
END

CLOSE EmpCursor
DEALLOCATE EmpCursor
```

What are the different Cursor Types?

There are various different cursor types, each of which has their own functional and performance characteristics. Depending on the needs of your application, you might choose any of these cursor types. Also, depending on whether you choose to use SQL-92 or Transact-SQL extended syntax you have different choices. Let's look at the various cursor types.

INSENSITIVE (SQL-92 Syntax)
The insensitive cursor executes the query and stores the entire set of data that is used for the cursor in tempdb. Thus, the query is only run once and the data is static. The result of this is good performance, since the cursor query is only run once, but no modifications to the cursor can be made, nor are changes made to the underlying tables reflected in the cursor. With an insensitive cursor, only FETCH NEXT is supported.

SCROLL (SQL-92 Syntax)
Unlike the insensitive cursor, the scroll cursor does support modifications as well as all cursor directional options such as FETCH NEXT, FETCH FIRST, FETCH LAST, FETCH PRIOR, FETCH RELATIVE and FETCH ABSOLUTE. In addition, changes to the underlying tables are reflected in the cursor. The downside is that the cursor is reevaluated on each call.

FORWARD_ONLY
The forward only cursor only supports traversing the cursor from the first row to the last in a forward direction, thus FETCH NEXT is the only fetch type allowed. Even though forward only cursors support uni-directional movement, the cursor is still evaluated each time a fetch is called, unless STATIC or KEYSET is specified.

STATIC
The STATIC option specifies that a temporary copy of the data is created and used by the cursor. This eliminates the re-evaluation of the cursor on each call, thus improving performance. This cursor type does not allow modifications, nor are changes to the underlying tables reflected in the subsequent fetch calls.

KEYSET
With the KEYSET cursor, the list of rows in the cursor are built into a table in tempdb and is set, however changes to nonkey values are reflected in subsequent fetch calls. However, inserts into the underlying tables are not reflected in the subsequent cursor calls. With keyset cursors, the underlying SELECT statement is not re-evaluated..

DYNAMIC
The dynamic cursor reflects all changes made to the underlying tables. Each time a fetch is called the cursor is re-evaluated. Thus if the SELECT statement that defines the cursor is time and resource consuming, performance will be degraded.

FAST_FORWARD
The fast forward cursor specifies a read only, forward only cursor that has performance enhancements enabled and does not re-evaluate the cursor on each fetch. This is good for performance if it is acceptable programmatically.

There are other cursor options that are available, but they only affect the functionality of the cursor, not performance. As you can see, there are numerous options, each with their own characteristics and performance.

Summary

This paper has outlined the different types of cursors available within SQL Server as well as the performance implications of those cursors types. By choosing the right cursor type based on your needs you can greatly improve the overall performance of your system, especially when cursors are used extensively throughout your application. By choosing a cursor type that does not re-evaluate the underlying query, resources and time can be saved.

Often you can tell that there is excessive cursor usage by profiling the system and looking for excessive resources being used for the same cursor call. These calls may present themselves as the SELECT statement itself or as an sp_executesql command.

If possible, use static cursors, or keyset cursors. Only use dynamic cursors if it is needed for your application. I have frequently seen applications that have defined dynamic cursors that never take advantage of the features of dynamic cursors. If you don't need it, don't use it. The results of performance on the cursor itself and on the system as a whole can be dramatic.

About the Author

Edward Whalen is vice president and principle consultant at Performance Tuning Corporation (www.perftuning.com). Performance Tuning Corporation provides database performance tuning, load testing and troubleshooting services on Oracle and MS SQL Server. Edward Whalen was a co-author on for SQL Server books from Microsoft Press;
     SQL Server 7 Administrator's Companion,
     SQL Server 7 Performance Tuning Technical Reference,
     SQL Server 2000 Administrator's Companion
     SQL Server 2000 Performance Tuning Technical Reference
Edward Whalen has also authored four Oracle books;
     Oracle Performance Tuning and Optimization (Oracle7)
     Teach Yourself Oracle8 in 21 Days
     Oracle Performance Tuning (Oracle8/9i)
     Teach Yourself Oracle9i in 21 Days (2002/2003)
Edward Whalen is considered a leader in database performance tuning.