Cursors

In SQL, the typical way to deal with data is by set-at-a-time. You've seen this over and over again in the examples, because you seldom use more than a statement or two to manipulate data and never any classic looping code: `for` loops, `while` loops, etc. However, it isn't always possible to deal with data in a set-based manner. **Cursors** are a T-SQL mechanism that allow you to get to an individual row and perform an action on it, rather than operating on a preselected group of rows.

Cursors have a tendency to draw the ire of administrators, for a couple of good reasons:

- First, they tend to lead to poor implementations of solutions to problems that are better resolved using set-based SQL. In these cases, the performance of a query using cursors will frequently be an order of magnitude slower than when using set-based operations.
- Unless you use the proper cursor types, you'll end up placing locks around your database for each row that you access. When using T-SQL cursors, this may be less of a problem because your T-SQL code will never have to wait for user interaction, but the problem certainly still exists.

The second of these reasons is really quite true, as there are occasions when you do have to use them. With the advent of user-defined functions, however, there is really only one situation in which the cursor provides the only possibility for implementation–when you need to perform some external command on each row in a set. For instance, a cursor would allow you to perform a custom stored procedure, containing business logic, on every row in a table.

Coding Cursors

Let's now discuss the basic syntax of cursors. To employ a cursor, you need to use the `SET` command.

```
SET @<cursorName> = CURSOR
      [ FORWARD_ONLY | SCROLL ]
      [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
      [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
      [ TYPE_WARNING ]
FOR <SELECT statement>
      [ FOR UPDATE [ OF <column1>, <column2>,...,<columnN> ] ]
```

> *If you have used cursors in the past, you're probably more familiar with the* `DECLARE CURSOR` *syntax. It's my opinion that using* `SET` *to set the cursor variable is a more coherent way of dealing with cursors. It leads to better code, because all cursors built this way are always local cursors and are dealt with in much the same manner as typical SQL Server variables. Because they are scoped to a particular batch, they don't have to be allocated and deallocated.*

Name

```
SET @<cursorName> = CURSOR
```

This declares a variable as a cursor data type. This may then be passed around to different procedures as a parameter. It has the same scope as a typical SQL variable, and this alleviates the need to explicitly close and deallocate cursors.

Direction

```
      [ FORWARD_ONLY | SCROLL ]
```

It generally benefits you to use a `FORWARD_ONLY` cursor, unless you have some important reason to move back and forth in the table (`SCROLL`). Forward-only cursors use fewer resources than scrolling ones as, once you've moved past the row, it can be discarded.

You'll usually leave this setting blank, because the `FAST_FORWARD` setting from the next set of optional parameters won't work with either `FORWARD_ONLY` or `SCROLL`, and `FAST_FORWARD` builds the most optimized cursor for typical use.

Cursor Type

```
      [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
```

With the `STATIC` option, SQL Server copies the data you need into its `tempdb` database, and then you can move about in the set. No changes made to the underlying data will be seen in the cursor. This is the default.

KEYSET cursors simply make a copy of the keys in tempdb and then, as you request a row, SQL Server goes out to the physical table and fetches the data. You can't see any new data, but you'll see any changes to existing data, including deletes.

DYNAMIC cursors are a bit more costly in terms of performance as the cursor values aren't materialized into a working set until you actually fetch data from them. With KEYSET cursors, the members of the table are fixed at the time of the cursor's creation. DYNAMIC cursors have their members determined only as the cursor is moved.

The best choice is usually to specify a FAST_FORWARD cursor, but this setting precludes any of the direction settings and doesn't allow scrolling. This isn't a static cursor because, if you make any modifications to the data, they will show up. In one of the examples shortly, you'll see how this property can affect the cursors you use.

Updatability

```
              [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
```

The READ_ONLY option should be obvious, but the other two are less so. They both allow you to update the data in the cursor. However, it's generally not a great idea to make use of the positional updates in T-SQL code, as it can be really confusing to understand exactly where you are in the set.

If you really need to make updates, use an enhancement to the UPDATE statement with a nonstatic cursor type (KEYSET or DYNAMIC) as follows:

```
UPDATE <tableName>
SET <field> = value
FROM <tableName>
WHERE CURRENT OF @<cursorVariableName>
```

The CURRENT OF keyword tells the statement to modify the row that the cursor is currently positioned on. You can also do DELETE operations as well.

To carry out updates in the cursor, you can use the SCROLL_LOCKS and OPTIMISTIC settings. SCROLL_LOCKS will guarantee that your updates and deletes (made in the position specified by CURRENT OF) succeed by placing locks on the underlying tables. OPTIMISTIC uses the table's timestamp column or, if none is available, builds a checksum for each row. This checksum is used in just the same way as the optimistic lock I've discussed before, by recalculating prior to making a change to see if another user has modified the row.

However, unless you find a really good reason for updating data within a cursor, don't try it. It's much easier to do the same task using code like this:

```
UPDATE <tableName>
SET <field> = value
FROM <tableName>
WHERE <pkey> = @<variable 'fetched into' using cursor>
```

This does almost the same thing as the `CURRENT OF` cursor and is easier to read and debug, because you can print out the variable as you work through the examples.

Cursor Type Downgrading

```
      [TYPE_WARNING]
```

If the `SELECT` statement that you've specified is unable to use a given cursor type, this setting requests an error message from SQL Server warning of this. Such a situation might arise if you try to use a `KEYSET` cursor on a set with no unique key with which to build the key set–SQL Server might choose to use a static cursor instead.

Cursor Performance

This example illustrates some of the performance issues with cursors. You seek to find out which stored procedures and functions in the `master` database have two or three parameters (as you did with the temporary table example), based on the `PARAMETERS` view of the `INFORMATION_SCHEMA`. You'll time the operation as it runs on a Dell 400MHz Pentium II laptop, with 256MB of RAM. First, you create a temporary table using a cursor as shown:

```
      SELECT getdate()               -- get time before the code is run

      --temporary table to hold the procedures as you loop through
      CREATE TABLE #holdProcedures
      (
         specific_name sysname
      )

      DECLARE @cursor CURSOR,        --cursor variable
              @c_routineName sysname     -- variable to hold the values as you loop

      SET @cursor = CURSOR FAST_FORWARD FOR SELECT DISTINCT specific_name
      FROM information_schema.parameters
      OPEN @cursor                    --activate the cursor
      FETCH NEXT FROM @cursor INTO @c_routineName  --get the first row

      WHILE @@fetch_status = 0       --this means that the row was fetched cleanly
         BEGIN
            --check to see if the count of parameters for this specific name
            --is between 2 and 3
            IF ( SELECT count(*)
                  FROM information_schema.parameters as parameters
                  WHERE parameters.specific_name = @c_routineName )
                     BETWEEN 2 AND 3
               BEGIN                  --it was, so put into temp storage
                  INSERT INTO #holdProcedures
                  VALUES (@c_routineName)
               END

            FETCH NEXT FROM @cursor INTO @c_routineName   --get the next row
         END
```

and once you're done, you can count the number of records as follows:

```
SELECT count(*)
FROM #holdProcedures

SELECT getdate() --output the time after the code is complete
```

Executing this statement, you get the following result:

```
-----------------------------------------------
2001-01-10 23:46:59.477

-----------------------------------------------
355

-----------------------------------------------
2001-01-10 23:47:00.917
```

This indicates that the operation took around 1.5 seconds to execute. This isn't too slow, but there is certainly a better way of doing this, as shown here:

```
SELECT GETDATE()    --again for demo purposes

SELECT COUNT(*)
--use a nested subquery to get all of the functions with
--the proper count of parameters
FROM (SELECT parameters.specific_name
      FROM information_schema.parameters AS parameters
      GROUP BY parameters.specific_name
      HAVING COUNT(*) BETWEEN 2 AND 3) AS counts

SELECT GETDATE()
```

This time you achieve a somewhat better result.

```
-----------------------------------------------
2001-01-10 23:50:25.160

-----------------------------------------------
355

-----------------------------------------------
2001-01-10 23:50:25.823
```

It takes 0.7 seconds to run the query now, which is about half the time taken to do the same task using cursors. Additionally, the code is far less verbose and easier for SQL Server to optimize.

Hopefully, this goes some way to demonstrating that cursors cause large performance problems, requiring many more resources than set-based operations.

Summary

Use cursors as little as possible. They have very large overhead, and are much slower than any of the set-based operations. However, when cursors really are needed, it's great to have them around. Just be certain to use them only as needed, and not as a crutch for poor programming habits.

I discuss some of the uses of cursors in Chapter 13.