CHAPTER 23

Row-by-Row Processing

It is common to find database applications that use cursors to process one row at a time. Developers tend to think about processing data in a row-by-row fashion. Oracle even uses something called *cursors* as a high-speed data access mechanism. Cursors in SQL Server are different. Because data manipulation through a cursor in SQL Server incurs significant additional overhead, database applications should avoid using cursors. T-SQL and SQL Server are designed to work best with sets of data, not one row at a time. Jeff Moden famously termed this type of processing RBAR (pronounced, "ree-bar"), meaning row by agonizing row. However, if a cursor must be used, then use a cursor with the least cost.

In this chapter, I cover the following topics:

- The fundamentals of cursors
- A cost analysis of different characteristics of cursors
- The benefits and drawbacks of a default result set over cursors
- Recommendations to minimize the cost overhead of cursors

Cursor Fundamentals

When a query is executed by an application, SQL Server returns a set of data consisting of rows. Generally, applications can't process multiple rows together; instead, they process one row at a time by walking through the result set returned by SQL Server. This functionality is provided by a *cursor*, which is a mechanism to work with one row at a time out of a multirow result set.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_23

T-SQL cursor processing usually involves the following steps:

1. Declare the cursor to associate it with a SELECT statement and define the characteristics of the cursor.
2. Open the cursor to access the result set returned by the SELECT statement.
3. Retrieve a row from the cursor. Optionally, modify the row through the cursor.
4. Move to additional rows in the result set.
5. Once all the rows in the result set are processed, close the cursor and release the resources assigned to the cursor.

You can create cursors using T-SQL statements or the data access layers used to connect to SQL Server. Cursors created using data access layers are commonly referred to as *client* cursors. Cursors written in T-SQL are referred to as *server* cursors. The following is an example of a server cursor processing query results from a table:

```
--Associate a SELECT statement to a cursor and define the
--cursor's characteristics
USE AdventureWorks2017;
GO
SET NOCOUNT ON
DECLARE MyCursor CURSOR /*<cursor characteristics>*/
FOR
SELECT adt.AddressTypeID,
       adt.Name,
       adt.ModifiedDate
FROM Person.AddressType AS adt;

--Open the cursor to access the result set returned by the
--SELECT statement
OPEN MyCursor;

--Retrieve one row at a time from the result set returned by
--the SELECT statement
```

```
DECLARE @AddressTypeId INT,
        @Name VARCHAR(50),
        @ModifiedDate DATETIME;

FETCH NEXT FROM MyCursor
INTO @AddressTypeId,
     @Name,
     @ModifiedDate;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'NAME =   ' + @Name;

    --Optionally, modify the row through the cursor
    UPDATE Person.AddressType
    SET Name = Name + 'z'
    WHERE CURRENT OF MyCursor;

    --Move through to additional rows in the data set
    FETCH NEXT FROM MyCursor
    INTO @AddressTypeId,
         @Name,
         @ModifiedDate;
END

--Close the cursor and release all resources assigned to the
--cursor
CLOSE MyCursor;
DEALLOCATE MyCursor;
```

Part of the overhead of the cursor depends on the cursor characteristics. The characteristics of the cursors provided by SQL Server and the data access layers can be broadly classified into three categories.

- *Cursor location*: Defines the location of the cursor creation
- *Cursor concurrency*: Defines the degree of isolation and synchronization of a cursor with the underlying content
- *Cursor type*: Defines the specific characteristics of a cursor

Before looking at the costs of cursors, I'll take a few pages to introduce the various characteristics of cursors. You can undo the changes to the `Person.AddressType` table with this query:

```
UPDATE Person.AddressType
SET Name = LEFT(Name, LEN(Name) - 1);
```

Cursor Location

Based on the location of its creation, a cursor can be classified into one of two categories.

- Client-side cursors
- Server-side cursors

The T-SQL cursors are always created on SQL Server. However, the database API cursors can be created on either the client or server side.

Client-Side Cursors

As its name signifies, a *client-side cursor is* created on the machine running the application, whether the app is a service, a data access layer, or the front end for the user. It has the following characteristics:

- It is created on the client machine.
- The cursor metadata is maintained on the client machine.
- It is created using the data access layers.
- It works against most of the data access layers (OLEDB providers and ODBC drivers).
- It can be a forward-only or static cursor.

Note Cursor types, including forward-only and static cursor types, are described later in the chapter in the "Cursor Types" section.

Server-Side Cursors

A *server-side cursor* is created on the SQL Server machine. It has the following characteristics:

- It is created on the server machine.
- The cursor metadata is maintained on the server machine.
- It is created using either data access layers or T-SQL statements.
- A server-side cursor created using T-SQL statements is tightly integrated with SQL Server.
- It can be any type of cursor. (Cursor types are explained later in the chapter.)

Note The cost comparison between client-side and server-side cursors is covered later in the chapter in the "Cost Comparison on Cursor Type" section.

Cursor Concurrency

Depending on the required degree of isolation and synchronization with the underlying content, cursors can be classified into the following concurrency models:

- *Read-only*: A nonupdatable cursor
- *Optimistic*: An updatable cursor that uses the optimistic concurrency model (no locks retained on the underlying data rows)
- *Scroll locks*: An updatable cursor that holds a lock on any data row to be updated

Read-Only

A read-only cursor is nonupdatable; no locks are held on the base tables. While fetching a cursor row, whether an (S) lock will be acquired on the underlying row depends upon the isolation level of the connection and any locking hints used in the `SELECT` statement

for the cursor. However, once the row is fetched, by default the locks are released. The following T-SQL statement creates a read-only T-SQL cursor:

```
DECLARE MyCursor CURSOR READ_ONLY FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Using as minimal locking overhead as possible makes the read-only type of cursor faster and safer. Just remember that you cannot manipulate data through the read-only cursor, which is the sacrifice you make for improved performance.

Optimistic

The optimistic with values concurrency model makes a cursor updatable. No locks are held on the underlying data. The factors governing whether an (S) lock will be acquired on the underlying row are the same as for a read-only cursor.

The optimistic concurrency model uses row versioning to determine whether a row has been modified since it was read into the cursor, instead of locking the row while it is read into the cursor. Version-based optimistic concurrency requires a `ROWVERSION` column in the underlying user table on which the cursor is created. The `ROWVERSION` data type is a binary number that indicates the relative sequence of modifications on a row. Each time a row with a `ROWVERSION` column is modified, SQL Server stores the current value of the global `ROWVERSION` value, `@@DBTS`, in the `ROWVERSION` column; it then increments the `@@DBTS` value.

Before applying a modification through the optimistic cursor, SQL Server determines whether the current `ROWVERSION` column value for the row matches the `ROWVERSION` column value for the row when it was read into the cursor. The underlying row is modified only if the `ROWVERSION` values match, indicating that the row hasn't been modified by another user in the meantime. Otherwise, an error is raised. In case of an error, refresh the cursor with the updated data.

If the underlying table doesn't contain a `ROWVERSION` column, then the cursor defaults to value-based optimistic concurrency, which requires matching the current value of the row with the value when the row was read into the cursor. The version-based concurrency control is more efficient than the value-based concurrency control since it requires less processing to determine the modification of the underlying row. Therefore,

for the best performance of a cursor with the optimistic concurrency model, ensure that the underlying table has a ROWVERSION column.

The following T-SQL statement creates an optimistic T-SQL cursor:

```
DECLARE MyCursor CURSOR OPTIMISTIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

A cursor with scroll locks concurrency holds a (U) lock on the underlying row until another cursor row is fetched or the cursor is closed. This prevents other users from modifying the underlying row when the cursor fetches it. The scroll locks concurrency model makes the cursor updatable.

The following T-SQL statement creates a T-SQL cursor with the scroll locks concurrency model:

```
DECLARE MyCursor CURSOR SCROLL_LOCKS FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Since locks are held on a row being referenced (until another cursor row is fetched or the cursor is closed), it blocks all the other users trying to modify the row during that period. This hurts database concurrency but ensures that you won't get errors if you're modifying data through the cursor.

Cursor Types

Cursors can be classified into the following four types:

- Forward-only cursors
- Static cursors
- Keyset-driven cursors
- Dynamic cursors

Let's take a closer look at these four types in the sections that follow.

Forward-Only Cursors

These are the characteristics of forward-only cursors:

- They operate directly on the base tables.
- Rows from the underlying tables are usually not retrieved until the cursor rows are fetched using the cursor `FETCH` operation. However, the database API forward-only cursor type, with the following additional characteristics, retrieves all the rows from the underlying table first:
  - Client-side cursor location
  - Server-side cursor location and read-only cursor concurrency
- They support forward scrolling only (`FETCH NEXT`) through the cursor.
- They allow all changes (`INSERT`, `UPDATE`, and `DELETE`) through the cursor. Also, these cursors reflect all changes made to the underlying tables.

The forward-only characteristic is implemented differently by the database API cursors and the T-SQL cursor. The data access layers implement the forward-only cursor characteristic as one of the four previously listed cursor types. But the T-SQL cursor doesn't implement the forward-only cursor characteristic as a cursor type; rather, it implements it as a property that defines the scrollable behavior of the cursor. Thus, for a T-SQL cursor, the forward-only characteristic can be used to define the scrollable behavior of one of the remaining three cursor types.

The T-SQL syntax provides a specific cursor type option, `FAST_FORWARD`, to create a fast-forward-only cursor. The nickname for the `FAST_FORWARD` cursor is the *fire hose* because it is the fastest way to move data through a cursor and because all the information flows one way. However, don't be surprised when the "firehose" is still not as fast as traditional set-based operations. The following T-SQL statement creates a fast-forward-only T-SQL cursor:

```
DECLARE MyCursor CURSOR FAST_FORWARD FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The FAST_FORWARD property specifies a forward-only, read-only cursor with performance optimizations enabled.

Static Cursors

These are the characteristics of static cursors:

- They create a snapshot of cursor results in the tempdb database when the cursor is opened. Thereafter, static cursors operate on the snapshot in the tempdb database.
- Data is retrieved from the underlying tables when the cursor is opened.
- Static cursors support all scrolling options: FETCH FIRST, FETCH NEXT, FETCH PRIOR, FETCH LAST, FETCH ABSOLUTE n, and FETCH RELATIVE n.
- Static cursors are always read-only; data modifications are not allowed through static cursors. Also, changes (INSERT, UPDATE, and DELETE) made to the underlying tables are not reflected in the cursor.

The following T-SQL statement creates a static T-SQL cursor:

```
DECLARE MyCursor CURSOR STATIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Some tests show that a static cursor can perform as well as—and sometimes faster than—a forward-only cursor. Be sure to test this behavior on your own system in situations where you must use a cursor.

Keyset-Driven Cursors

These are the characteristics of keyset-driven cursors:

- Keyset cursors are controlled by a set of unique identifiers (or keys) known as a *keyset*. The keyset is built from a set of columns that uniquely identify the rows in the result set.
- These cursors create the keyset of rows in the tempdb database when the cursor is opened.

- Membership of rows in the cursor is limited to the keyset of rows created in the tempdb database when the cursor is opened.
- On fetching a cursor row, the database engine first looks at the keyset of rows in tempdb and then navigates to the corresponding data row in the underlying tables to retrieve the remaining columns.
- They support all scrolling options.
- Keyset cursors allow all changes through the cursor. An `INSERT` performed outside the cursor is not reflected in the cursor since the membership of rows in the cursor is limited to the keyset of rows created in the tempdb database on opening the cursor. An `INSERT` through the cursor appears at the end of the cursor. A `DELETE` performed on the underlying tables raises an error when the cursor navigation reaches the deleted row. An `UPDATE` on the nonkeyset columns of the underlying tables is reflected in the cursor. An `UPDATE` on the keyset columns is treated like a `DELETE` of an old key value and the `INSERT` of a new key value. If a change disqualifies a row for membership or affects the order of a row, then the row does not disappear or move unless the cursor is closed and reopened.

The following T-SQL statement creates a keyset-driven T-SQL cursor:

```
DECLARE MyCursor CURSOR KEYSET FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Dynamic Cursors

These are the characteristics of dynamic cursors:

- Dynamic cursors operate directly on the base tables.
- The membership of rows in the cursor is not fixed since they operate directly on the base tables.
- As with forward-only cursors, rows from the underlying tables are not retrieved until the cursor rows are fetched using a cursor `FETCH` operation.

- Dynamic cursors support all scrolling options except `FETCH ABSOLUTE n`, since the membership of rows in the cursor is not fixed.
- These cursors allow all changes through the cursor. Also, all changes made to the underlying tables are reflected in the cursor.
- Dynamic cursors don't support all properties and methods implemented by the database API cursors. Properties such as `AbsolutePosition`, `Bookmark`, and `RecordCount`, as well as methods such as `clone` and `Resync`, are not supported by dynamic cursors. Instead, they are supported by keyset-driven cursors.

The following T-SQL statement creates a dynamic T-SQL cursor:

```
DECLARE MyCursor CURSOR DYNAMIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The dynamic cursor is absolutely the slowest possible cursor to use in all situations. It takes more locks and holds them longer, which radically increases its poor performance. Take this into account when designing your system.

Cursor Cost Comparison

Now that you've seen the different cursor flavors, let's look at their costs. If you must use a cursor, you should always use the lightest-weight cursor that meets the requirements of your application. The cost comparisons among the different characteristics of the cursors are detailed next.

Cost Comparison on Cursor Location

The client-side and server-side cursors have their own cost benefits and overhead, as explained in the sections that follow.

Client-Side Cursors

Client-side cursors have the following cost benefits compared to server-side cursors:

- *Higher scalability*: Since the cursor metadata is maintained on the individual client machines connected to the server, the overhead of maintaining the cursor metadata is taken up by the client machines. Consequently, the ability to serve a larger number of users is not limited by the server resources.
- *Fewer network round-trips*: Since the result set returned by the `SELECT` statement is passed to the client where the cursor is maintained, extra network round-trips to the server are not required while retrieving rows from the cursor.
- *Faster scrolling*: Since the cursor is maintained locally on the client machine, it's potentially faster to walk through the rows of the cursor.
- *Highly portable*: Since the cursor is implemented using data access layers, it works across a large range of databases: SQL Server, Oracle, Sybase, and so forth.

Client-side cursors have the following cost overhead or drawbacks:

- *Higher pressure on client resources*: Since the cursor is managed at the client side, it increases pressure on the client resources. But it may not be all that bad, considering that most of the time the client applications are web applications and scaling out web applications (or web servers) is quite easy using standard load-balancing solutions. On the other hand, scaling out a transactional SQL Server database is still an art!
- *Support for limited cursor types*: Dynamic and keyset-driven cursors are not supported.

- *Only one active cursor-based statement on one connection*: As many rows of the result set as the client network can buffer are arranged in the form of network packets and sent to the client application. Therefore, until all the cursor's rows are fetched by the application, the database connection remains busy, pushing the rows to the client. During this period, other cursor-based statements cannot use the connection. This is negated by taking advantage of multiple active result sets (MARS), which would allow a connection to have a second active cursor.

Server-Side Cursors

Server-side cursors have the following cost benefits:

- *Multiple active cursor-based statements on one connection*: While using server-side cursors, no results are left outstanding on the connection between the cursor operations. This frees the connection, allowing the use of multiple cursor-based statements on one connection at the same time. In the case of client-side cursors, as explained previously, the connection remains busy until all the cursor rows are fetched by the application. This means they cannot be used simultaneously by multiple cursor-based statements.
- *Row processing near the data*: If the row processing involves joining with other tables and a considerable amount of set operations, then it is advantageous to perform the row processing near the data using a server-side cursor.
- *Less pressure on client resources*: It reduces pressure on the client resources. But this may not be that desirable because, if the server resources are maxed out (instead of the client resources), then it will require scaling out the database, which is a difficult proposition.
- *Support for all cursor types*: Client-side cursors have limitations on which types of cursors can be supported. There are no limits on the server-side cursors.

Server-side cursors have the following cost overhead or disadvantages:

- *Lower scalability*: They make the server less scalable since server resources are consumed to manage the cursor.
- *More network round-trips*: They increase network round-trips if the cursor row processing is done in the client application. The number of network round-trips can be optimized by processing the cursor rows in the stored procedure or by using the cache size feature of the data access layer.
- *Less portable*: Server-side cursors implemented using T-SQL cursors are not readily portable to other databases because the syntax of the database code managing the cursor is different across databases.

Cost Comparison on Cursor Concurrency

As expected, cursors with a higher concurrency model create the least amount of blocking in the database and support higher scalability, as explained in the following sections.

Read-Only

The read-only concurrency model provides the following cost benefits:

- *Lowest locking overhead*: The read-only concurrency model introduces the least locking and synchronization overhead on the database. Since (S) locks are not held on the underlying row after a cursor row is fetched, other users are not blocked from accessing the row. Furthermore, the (S) lock acquired on the underlying row while fetching the cursor row can be avoided by using the `NO_LOCK` locking hint in the `SELECT` statement of the cursor, but only if you don't care about what kind of data you get back because of dirty reads.
- *Highest concurrency*: Since additional locks are not held on the underlying rows, the read-only cursor doesn't block other users from accessing the underlying tables. The shared lock is still acquired.

The main drawback of the read-only cursor is as follows:

- *Nonupdatable*: The content of underlying tables cannot be modified through the cursor.

Optimistic

The optimistic concurrency model provides the following benefits:

- *Low locking overhead*: Similar to the read-only model, the optimistic concurrency model doesn't hold an (S) lock on the cursor row after the row is fetched. To further improve concurrency, the `NOLOCK` locking hint can also be used, as in the case of the read-only concurrency model. But, please know that `NOLOCK` can absolutely lead to incorrect data or missing or extra rows, so its use requires careful planning. Modification through the cursor to an underlying row requires exclusive rights on the row as required by an action query.
- *High concurrency*: Since only a shared lock is used on the underlying rows, the cursor doesn't block other users from accessing the underlying tables. But the modification through the cursor to an underlying row will block other users from accessing the row during the modification.

The following examples detail the cost overhead of the optimistic concurrency model:

- *Row versioning*: Since the optimistic concurrency model allows the cursor to be updatable, an additional cost is incurred to ensure that the current underlying row is first compared (using either version-based or value-based concurrency control) with the original cursor row fetched before applying a modification through the cursor. This prevents the modification through the cursor from accidentally overwriting the modification made by another user after the cursor row is fetched.

- *Concurrency control without a ROWVERSION column*: As explained previously, a `ROWVERSION` column in the underlying table allows the cursor to perform an efficient version-based concurrency control. In case the underlying table doesn't contain a `ROWVERSION` column, the cursor resorts to value-based concurrency control, which requires matching the current value of the row to the value when the row was read into the cursor. This increases the cost of the concurrency control. Both forms of concurrency control will cause additional overhead in tempdb.

Scroll Locks

The major benefit of the scroll locks concurrency model is as follows:

- *Simple concurrency control*: By locking the underlying row corresponding to the last fetched row from the cursor, the cursor assures that the underlying row can't be modified by another user. This eliminates the versioning overhead of optimistic locking. Also, since the row cannot be modified by another user, the application is relieved from checking for a row-mismatch error.

The scroll locks concurrency model incurs the following cost overhead:

- *Highest locking overhead*: The scroll locks concurrency model introduces a pessimistic locking characteristic. A (U) lock is held on the last cursor row fetched, until another cursor row is fetched or the cursor is closed.
- *Lowest concurrency*: Since a (U) lock is held on the underlying row, all other users requesting a (U) or an (X) lock on the underlying row will be blocked. This can significantly hurt concurrency. Therefore, please avoid using this cursor concurrency model unless absolutely necessary.

Cost Comparison on Cursor Type

Each of the basic four cursor types mentioned in the "Cursor Fundamentals" section earlier in the chapter incurs a different cost overhead on the server. Choosing an incorrect cursor type can hurt database performance. Besides the four basic cursor types, a fast-forward-only cursor (a variation of the forward-only cursor) is provided to enhance performance. The cost overhead of these cursor types is explained in the sections that follow.

Forward-Only Cursors

These are the cost benefits of forward-only cursors:

- *Lower cursor open cost than static and keyset-driven cursors*: Since the cursor rows are not retrieved from the underlying tables and are not copied into the tempdb database during cursor open, the forward-only T-SQL cursor opens quickly. Similarly, the forward-only, server-side API cursors with optimistic/scroll locks concurrency open quickly since they do not retrieve the rows during cursor open.
- *Lower scroll overhead*: Since only `FETCH NEXT` can be performed on this cursor type, it requires less overhead to support different scroll operations.
- *Lower impact on the tempdb database than static and keyset-driven cursors*: Since the forward-only T-SQL cursor doesn't copy the rows from the underlying tables into the tempdb database, no additional pressure is created on the database.

The forward-only cursor type has the following drawbacks:

- *Lower concurrency*: Every time a cursor row is fetched, the corresponding underlying row is accessed with a lock request depending on the cursor concurrency model (as noted earlier in the discussion about concurrency). It can block other users from accessing the resource.
- *No backward scrolling*: Applications requiring two-way scrolling can't use this cursor type. But if the applications are designed properly, then it isn't difficult to live without backward scrolling.

Fast-Forward-Only Cursor

The fast-forward-only cursor is the fastest and least expensive cursor type. This forward-only and read-only cursor is specially optimized for performance. Because of this, you should always prefer it to the other SQL Server cursor types.

Furthermore, the data access layer provides a fast-forward-only cursor on the client side. That type of cursor uses a so-called *default result set* to make cursor overhead almost disappear.

Note The default result set is explained later in the chapter in the "Default Result Set" section.

Static Cursors

These are the cost benefits of static cursors:

- *Lower fetch cost than other cursor types*: Since a snapshot is created in the tempdb database from the underlying rows on opening the cursor, the cursor row fetch is targeted to the snapshot instead of the underlying rows. This avoids the lock overhead that would otherwise be required to fetch the cursor rows.
- *No blocking on underlying rows*: Since the snapshot is created in the tempdb database, other users trying to access the underlying rows are not blocked.

On the downside, the static cursor has the following cost overhead:

- *Higher open cost than other cursor types*: The cursor open operation of the static cursor is slower than that of other cursor types since all the rows of the result set have to be retrieved from the underlying tables and the snapshot has to be created in the tempdb database during the cursor open.
- *Higher impact on tempdb than other cursor types*: There can be significant impact on server resources for creating, populating, and cleaning up the snapshot in the tempdb database.

Keyset-Driven Cursors

These are the cost benefits of keyset-driven cursors:

- *Lower open cost than the static cursor*: Since only the keyset, not the complete snapshot, is created in the tempdb database, the keyset-driven cursor opens faster than the static cursor. SQL Server populates the keyset of a large keyset-driven cursor asynchronously, which shortens the time between when the cursor is opened and when the first cursor row is fetched.
- *Lower impact on tempdb than that with the static cursor*: Because the keyset-driven cursor is smaller, it uses less space in tempdb.

The cost overhead of keyset-driven cursors is as follows:

- *Higher open cost than forward-only and dynamic cursors*: Populating the keyset in the tempdb database makes the cursor open operation of the keyset-driven cursor costlier than that of forward-only (with the exceptions mentioned earlier) and dynamic cursors.
- *Higher fetch cost than other cursor types*: For every cursor row fetch, the key in the keyset has to be accessed first, and then the corresponding underlying row in the user database can be accessed. Accessing both the tempdb and the user database for every cursor row fetch makes the fetch operation costlier than that of other cursor types.
- *Higher impact on tempdb than forward-only and dynamic cursors*: Creating, populating, and cleaning up the keyset in tempdb impacts server resources.
- *Higher lock overhead and blocking than the static cursor*: Since row fetch from the cursor retrieves rows from the underlying table, it acquires an (S) lock on the underlying row (unless the `NOLOCK` locking hint is used) during the row fetch operation.

Dynamic Cursor

The dynamic cursor has the following cost benefits:

- *Lower open cost than static and keyset-driven cursors*: Since the cursor is opened directly on the underlying rows without copying anything to the tempdb database, the dynamic cursor opens faster than the static and keyset-driven cursors.
- *Lower impact on tempdb than static and keyset-driven cursors*: Since nothing is copied into tempdb, the dynamic cursor places far less strain on tempdb than the other cursor types.

The dynamic cursor has the following cost overhead:

- *Higher lock overhead and blocking than the static cursor*: Every cursor row fetch in a dynamic cursor requeries the underlying tables involved in the `SELECT` statement of the cursor. The dynamic fetches are generally expensive because the original select condition might have to be reexecuted.

For a summary of the different cursors, their positives and negatives, please refer to Table 23-1.

Table 23-1. *Comparing Cursors*

Cursor Type	Positives	Negatives
Forward-only	Lower cost, lower scroll overhead, lower impact on tempdb	Lower concurrency, no backward scrolling
Fast-forward-only	Fastest cursor, lowest cost, lowest impact	No backward scrolling, no concurrency
Static	Lower fetch cost, no blocking, forward and backward scrolling	Higher open cost, higher impact on tempdb, no concurrency
Keyset-driven	Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency	Higher open cost, highest fetch cost, highest impact on tempdb, higher locking costs
Dynamic	Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency	Highest locking costs