

Version

SQL Server 2019

Filter by title

- hierarchyid methods (database engine)
- Numeric
- String & binary
- Spatial geography & instances (geography Data Type)
- Spatial geometry & instances (geometry Data Type)
- Data types
- XML
- DBCC
- Functions
- Language elements
  - Language elements
    - General
    - Control-of-Flow
    - Cursors
    - Cursors
    - CLOSE
    - DEALLOCATE
    - DECLARE CURSOR**
    - FETCH
    - OPEN
  - Expressions
  - Operators
  - Transactions
  - Variables
  - Queries
  - Statements
  - Query

# DECLARE CURSOR (Transact-SQL)

Article • 11/30/2021 • 13 minutes to read • 12 contributors

Applies to: SQL Server (all supported versions) Azure SQL Database Azure SQL Managed Instance

Defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. **DECLARE CURSOR** accepts both a syntax based on the ISO standard and a syntax using a set of Transact-SQL extensions.

Transact-SQL Syntax Conventions

## Syntax

```
syntaxsql
ISO Syntax
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ_ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[;]
Transact-SQL extended syntax
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

**Note**  
To view Transact-SQL syntax for SQL Server 2014 and earlier, see [Previous versions documentation](#).

## Arguments

*cursor\_name*  
Is the name of the Transact-SQL server cursor defined. *cursor\_name* must conform to the rules for identifiers.

**INSENSITIVE**  
Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications. When ISO syntax is used, if **INSENSITIVE** is omitted, committed deletes and updates made to the underlying tables (by any user) are reflected in subsequent fetches.

**SCROLL**  
Specifies that all fetch options (**FIRST**, **LAST**, **PRIOR**, **NEXT**, **RELATIVE**, **ABSOLUTE**) are available. If **SCROLL** is not specified in an ISO **DECLARE CURSOR**, **NEXT** is the only fetch option supported. **SCROLL** cannot be specified if **FAST\_FORWARD** is also specified. If **SCROLL** is not specified then only the fetch option **NEXT** is available and the cursor becomes **FORWARD\_ONLY**.

*select\_statement*  
Is a standard **SELECT** statement that defines the result set of the cursor. The keywords **FOR BROWSE**, and **INTO** are not allowed within *select\_statement* of a cursor declaration.

SQL Server implicitly converts the cursor to another type if clauses in *select\_statement* conflict with the functionality of the requested cursor type.

**READ\_ONLY**  
Prevents updates made through this cursor. The cursor cannot be referenced in a **WHERE CURRENT OF** clause in an **UPDATE** or **DELETE** statement. This option overrides the default capability of a cursor to be updated.

*UPDATE [OF column\_name [,...n]]*  
Defines updatable columns within the cursor. If **OF <column\_name> [ , <...n>]** is specified, only the columns listed allow modifications. If **UPDATE** is specified without a column list, all columns can be updated.

*cursor\_name*  
Is the name of the Transact-SQL server cursor defined. *cursor\_name* must conform to the rules for identifiers.

**LOCAL**  
Specifies that the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is only valid within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure **OUTPUT** parameter. An **OUTPUT** parameter is used to pass the local cursor back to the calling batch, stored procedure, or trigger, which can assign the parameter to a cursor-variable to reference the cursor after the stored procedure terminates. The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless the cursor was passed back in an **OUTPUT** parameter. If it is passed back in an **OUTPUT** parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.

**GLOBAL**  
Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is only implicitly deallocated at disconnect.

**Note**  
If neither **GLOBAL** or **LOCAL** is specified, the default is controlled by the setting of the default to local cursor database option.

**FORWARD\_ONLY**  
Specifies that the cursor can only move forward and be scrolled from the first to the last row. **FETCH NEXT** is the only supported fetch option. All insert, update, and delete statements made by the current user (or committed by other users) that affect rows in the result set are visible as the rows are fetched. Because the cursor cannot be scrolled backward, however, changes made to rows in the database after the row was fetched are not visible through the cursor. Forward-only cursors are dynamic by default, meaning that all changes are detected as the current row is processed. This provides faster cursor opening and enables the result set to display updates made to the underlying tables. While forward-only cursors do not support backward scrolling, applications can return to the beginning of the result set by closing and reopening the cursor. If **FORWARD\_ONLY** is specified without the **STATIC**, **KEYSET**, or **DYNAMIC** keywords, the cursor operates as a dynamic cursor. When neither **FORWARD\_ONLY** nor **SCROLL** is specified, **FORWARD\_ONLY** is the default, unless the keywords **STATIC**, **KEYSET**, or **DYNAMIC** are specified. **STATIC**, **KEYSET**, and **DYNAMIC** cursors default to **SCROLL**. Unlike database APIs such as ODBC and ADO, **FORWARD\_ONLY** is supported with **STATIC**, **KEYSET**, and **DYNAMIC** Transact-SQL cursors.

**STATIC**  
Specifies that the cursor always displays the result set as it was when the cursor was first opened, and makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**. Therefore inserts, updates, and deletes made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not detect changes made to the membership, order, or values of the result set after the cursor is opened. Static cursors may detect their own updates, deletes, and inserts, although they are not required to do so. For example, suppose a static cursor fetches a row, and another application then updates that row. If the application refetches the row from the static cursor, the values it sees are unchanged, despite the changes made by the other application. All types of scrolling are supported.

**KEYSET**  
Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into a table in **tempdb** known as the **keyset**. This cursor provides functionality between a static and a dynamic cursor in its ability to detect changes. Like a static cursor, it does not always detect changes to the membership and order of the result set. Like a dynamic cursor, it does detect changes to the values of rows in the result set. **Keyset-driven cursors** are controlled by a set of unique identifiers (keys) known as the **keyset**. The keys are built from a set of columns that uniquely identify the rows in the result set. The **keyset** is the set of key values from all the rows returned by the query statement. With **keyset-driven cursors**, a key is built and saved for each row in the cursor and stored either on the client workstation or on the server. When you access each row, the stored key is used to fetch the current data values from the data source. In a **keyset-driven cursor**, result set membership is frozen when the **keyset** is fully populated. Thereafter, additions or updates that affect membership are not a part of the result set until it is reopened. Changes to data values (made either by the **keyset** owner or other processes) are visible as the user scrolls through the result set:

- If a row is deleted, an attempt to fetch the row returns an **@FETCH\_STATUS** of -2 because the deleted row appears as a gap in the result set. The key for the row exists in the **keyset**, but the row no longer exists in the result set.
- Inserts made outside the cursor (by other processes) are visible only if the cursor is closed and reopened. Inserts made from inside the cursor are visible at the end of the result set.
- Updates of key values from outside the cursor resemble a delete of the old row followed by an insert of the new row. The row with the new values is not visible, and attempts to fetch the row with the old values return an **@FETCH\_STATUS** of -2. The new values are visible if the update is done through the cursor by specifying the **WHERE CURRENT OF** clause.

**Note**  
If the query references at least one table without a unique index, the **keyset** cursor is converted to a static cursor.

**DYNAMIC**  
Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor and fetch a new record, regardless of whether the changes occur from inside the cursor or by other users outside the cursor. Therefore all insert, update, and delete statements made by all users are visible through the cursor. The data values, order, and membership of the rows can change on each fetch. The **ABSOLUTE** fetch option is not supported with dynamic cursors. Updates made outside the cursor are not visible until they are committed (unless the cursor transaction isolation level is set to **UNCOMMITTED**). For example, suppose a dynamic cursor fetches two rows and another application then updates one of those rows and deletes the other. If the dynamic cursor then fetches those rows, it will not find the deleted row, but it will display the new values for the updated row.

**FAST\_FORWARD**  
Specifies a **FORWARD\_ONLY**, **READ\_ONLY** cursor with performance optimizations enabled. **FAST\_FORWARD** cannot be specified if **SCROLL** or **FOR\_UPDATE** is also specified. This type of cursor does not allow data modifications from inside the cursor.

**Note**  
Both **FAST\_FORWARD** and **FORWARD\_ONLY** can be used in the same **DECLARE CURSOR** statement.

**READ\_ONLY**  
Prevents updates made through this cursor. The cursor cannot be referenced in a **WHERE CURRENT OF** clause in an **UPDATE** or **DELETE** statement. This option overrides the default capability of a cursor to be updated.

**SCROLL\_LOCKS**  
Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. SQL Server locks the rows as they are read into the cursor to ensure their availability for later modifications. **SCROLL\_LOCKS** cannot be specified if **FAST\_FORWARD** or **STATIC** is also specified.

**OPTIMISTIC**  
Specifies that positioned updates or deletes made through the cursor do not succeed if the row has been updated since it was read into the cursor. SQL Server does not lock rows as they are read into the cursor. It instead uses comparisons of **timestamp** column values, or a checksum value if the table has no **timestamp** column, to determine whether the row was modified after it was read into the cursor. If the row was modified, the attempted positioned update or delete fails. **OPTIMISTIC** cannot be specified if **FAST\_FORWARD** is also specified.

**TYPE\_WARNING**  
Specifies that a warning message is sent to the client when the cursor is implicitly converted from the requested type to another.

*select\_statement*  
Is a standard **SELECT** statement that defines the result set of the cursor. The keywords **COMPUTE**, **COMPUTE BY**, **FOR BROWSE**, and **INTO** are not allowed within *select\_statement* of a cursor declaration.

**Note**  
You can use a query hint within a cursor declaration; however, if you also use the **FOR UPDATE OF** clause, specify **OPTION (<query\_hint>)** after **FOR UPDATE OF**.

SQL Server implicitly converts the cursor to another type if clauses in *select\_statement* conflict with the functionality of the requested cursor type. For more information, see [Implicit Cursor Conversions](#).

*FOR UPDATE [OF column\_name [,...n]]*  
Defines updatable columns within the cursor. If **OF <column\_name> [ , <...n>]** is supplied, only the columns listed allow modifications. If **UPDATE** is specified without a column list, all columns can be updated, unless the **READ\_ONLY** concurrency option was specified.

## Remarks

**DECLARE CURSOR** defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The **OPEN** statement populates the result set, and **FETCH** returns a row from the result set. The **CLOSE** statement releases the current result set associated with the cursor. The **DEALLOCATE** statement releases the resources used by the cursor.

The first form of the **DECLARE CURSOR** statement uses the ISO syntax for declaring cursor behaviors. The second form of **DECLARE CURSOR** uses Transact-SQL extensions that allow you to define cursors using the same cursor types used in the database API cursor functions of ODBC or ADO.

You cannot mix the two forms. If you specify the **SCROLL** or **INSENSITIVE** keywords before the **CURSOR** keyword, you cannot use any keywords between the **CURSOR** and **FOR <select\_statement>** keywords. If you specify any keywords between the **CURSOR** and **FOR <select\_statement>** keywords, you cannot specify **SCROLL** or **INSENSITIVE** before the **CURSOR** keyword.

If a **DECLARE CURSOR** using Transact-SQL syntax does not specify **READ\_ONLY**, **OPTIMISTIC**, or **SCROLL\_LOCKS**, the default is as follows:

- If the **SELECT** statement does not support updates (insufficient permissions, accessing remote tables that do not support updates, and so on), the cursor is **READ\_ONLY**.
- STATIC** and **FAST\_FORWARD** cursors default to **READ\_ONLY**.
- DYNAMIC** and **KEYSET** cursors default to **OPTIMISTIC**.

Cursor names can be referenced only by other Transact-SQL statements. They cannot be referenced by database API functions. For example, after declaring a cursor, the cursor name cannot be referenced from OLE DB, ODBC or ADO functions or methods. The cursor rows cannot be fetched using the fetch functions or methods of the APIs; the rows can be fetched only by Transact-SQL **FETCH** statements.

After a cursor has been declared, these system stored procedures can be used to determine the characteristics of the cursor.

System stored procedures	Description
sp_cursor_list	Returns a list of cursors currently visible on the connection and their attributes.
sp_describe_cursor	Describes the attributes of a cursor, such as whether it is a forward-only or scrolling cursor.
sp_describe_cursor_columns	Describes the attributes of the columns in the cursor result set.
sp_describe_cursor_tables	Describes the base tables accessed by the cursor.

Variables may be used as part of the *select\_statement* that declares a cursor. Cursor variable values do not change after a cursor is declared.

## Permissions

Permissions of **DECLARE CURSOR** default to any user that has **SELECT** permissions on the views, tables, and columns used in the cursor.

## Limitations and Restrictions

You cannot use cursors or triggers on a table with a clustered columnstore index. This restriction does not apply to nonclustered columnstore indexes; you can use cursors and triggers on a table with a nonclustered columnstore index.

## Examples

### A. Using simple cursor and syntax

The result set generated at the opening of this cursor includes all rows and all columns in the table. This cursor can be updated, and all updates and deletes are represented in fetches made against this cursor. **FETCH NEXT** is the only fetch available because the **SCROLL** option has not been specified.

```
SQL
DECLARE vend_cursor CURSOR
FOR SELECT * FROM Purchasing.Vendor
OPEN vend_cursor
FETCH NEXT FROM vend_cursor;
```

### B. Using nested cursors to produce report output

The following example shows how cursors can be nested to produce complex reports. The inner cursor is declared for each vendor.

```
SQL
SET NOCOUNT ON;

DECLARE @vendor_id INT, @vendor_name NVARCHAR(50),
@message VARCHAR(80), @product NVARCHAR(50);

PRINT '----- Vendor Products Report -----';

DECLARE vend_cursor CURSOR FOR
SELECT VendorID, Name
FROM Purchasing.Vendor
WHERE PreferredVendorStatus = 1
ORDER BY VendorID;

OPEN vend_cursor

FETCH NEXT FROM vend_cursor
INTO @vendor_id, @vendor_name

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT ' '
    SELECT @message = '----- Products From Vendor: ' +
@vendor_name
    PRINT @message

    -- Declare an inner cursor based
    -- on vendor_id from the outer cursor.

    DECLARE product_cursor CURSOR FOR
    SELECT v.Name
    FROM Purchasing.ProductVendor pv, Production.Product p
    WHERE pv.ProductID = v.ProductID AND
pv.VendorID = @vendor_id -- Variable value from the outer cursor

    OPEN product_cursor
    FETCH NEXT FROM product_cursor INTO @product

    IF @@FETCH_STATUS <= 0
    PRINT ' <<None>>'

    WHILE @@FETCH_STATUS = 0
    BEGIN
        SELECT @message = ' ' + @product
        PRINT @message
        FETCH NEXT FROM product_cursor INTO @product
        END

    CLOSE product_cursor
    DEALLOCATE product_cursor
    -- Dec the next vendor
    FETCH NEXT FROM vend_cursor
    INTO @vendor_id, @vendor_name
    END

    DEALLOCATE vend_cursor;
```

## See Also

- @@FETCH\_STATUS (Transact-SQL)
- CLOSE (Transact-SQL)
- Cursors (Transact-SQL)
- DEALLOCATE (Transact-SQL)
- FETCH (Transact-SQL)
- SELECT (Transact-SQL)
- sp\_configure (Transact-SQL)