

SQL has a set of statements known as Data Manipulation Language (DML) that deals with data manipulation. Some people think that DML involves only data modification, but it also involves data retrieval.

DML includes the statements *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *TRUNCATE*, and *MERGE*

Inserting data

T-SQL provides several statements for inserting data into tables:

1. *INSERT VALUES*
2. *INSERT SELECT*
3. *INSERT EXEC*
4. *SELECT INTO*
5. *BULK INSERT*

The ***INSERT VALUES*** statement

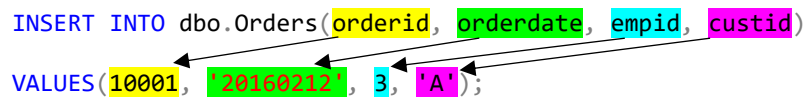
You use the standard *INSERT VALUES* statement to insert rows into a table based on specified values.

Run the following code to create the *Orders* table:

```
USE TSQLV4;
DROP TABLE IF EXISTS dbo.Orders;
CREATE TABLE dbo.Orders
(
  orderid INT NOT NULL
  CONSTRAINT PK_Orders PRIMARY KEY,
  orderdate DATE NOT NULL
  CONSTRAINT DFT_orderdate DEFAULT(SYSDATETIME()),
  empid INT NOT NULL,
  custid VARCHAR(10) NOT NULL
);
```

The following example demonstrates how to use the *INSERT VALUES* statement to insert a single row into the *Orders* table:

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
VALUES(10001, '20160212', 3, 'A');
```



If you don't specify a value for a column, Microsoft SQL Server will use a default value if one was defined for the column. If a default value isn't defined and the column allows *NULLs*, a *NULL* will be used. If no default is defined and the column does not allow *NULLs* and does not somehow get its value automatically, your *INSERT* statement will fail.

The following statement doesn't specify a value for the *orderdate* column; rather, it relies on its default (*SYSDATETIME*):

```
INSERT INTO dbo.Orders(orderid, empid, custid)
VALUES(10002, 5, 'B');
```

T-SQL supports an enhanced standard *VALUES* clause you can use to specify multiple rows separated by commas. For example, the following statement inserts four rows into the *Orders* table:

```
INSERT INTO dbo.Orders
(orderid, orderdate, empid, custid)
VALUES
(10003, '20160213', 4, 'B'),
(10004, '20160214', 1, 'A'),
(10005, '20160213', 1, 'C'),
(10006, '20160215', 3, 'C');
```

This statement is processed as a transaction, meaning that if any row fails to enter the table, none of the rows in the statement enters the table.

There's more to this enhanced *VALUES* clause. You can use it as a table-value constructor to construct a derived table. Here's an example:

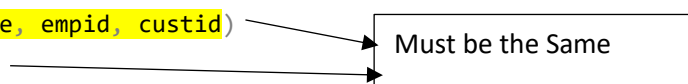
```
SELECT *
FROM ( VALUES
(10003, '20160213', 4, 'B'),
(10004, '20160214', 1, 'A'),
(10005, '20160213', 1, 'C'),
(10006, '20160215', 3, 'C') )
AS O(orderid, orderdate, empid, custid);
```

The ***INSERT SELECT*** statement

The standard *INSERT SELECT* statement inserts a set of rows returned by a *SELECT* query into a target table.

the following code inserts into the *dbo.Orders* table the result of a query against the *Sales.Orders* table and returns orders that were shipped to the United Kingdom:

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = N'UK';
```



Must be the Same

The ***INSERT EXEC*** statement

You use the *INSERT EXEC* statement to insert a result set returned from a stored procedure or a dynamic SQL batch into a target table. The *INSERT EXEC* statement is similar in syntax and concept to the *INSERT SELECT* statement, but instead of using a *SELECT* statement, you specify an *EXEC* statement.

For example, the following code creates a stored procedure called *Sales.GetOrders*, and it returns orders that were shipped to a specified input country (with the *@country* parameter):

```
DROP PROC IF EXISTS Sales.GetOrders;
GO
CREATE PROC Sales.GetOrders
@country AS NVARCHAR(40)
AS
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = @country;
GO
```

To test the stored procedure, execute it with the input country *France*:

➔ `EXEC Sales.GetOrders @country = N'France';`

By using an *INSERT EXEC* statement, you can insert the result set returned from the procedure into the *dbo.Orders* table:

➔ `INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
EXEC Sales.GetOrders @country = N'France';`

The ***SELECT INTO*** statement

T-SQL statement that creates a target table and populates it with the result set of a query.

```
DROP TABLE IF EXISTS dbo.Orders;
SELECT orderid, orderdate, empid, custid
INTO dbo.Orders
FROM Sales.Orders;
```

The **BULK INSERT** statement

You use the *BULK INSERT* statement to insert into an existing table data originating from a file. In the statement, you specify the target table, the source file, and options

```
BULK INSERT dbo.Orders FROM 'c:\temp\orders.txt'
WITH
(
  DATAFILETYPE = 'char',
  FIELDTERMINATOR = ',',
  ROWTERMINATOR = '\n'
);
```

The **Identity** property and the sequence object

SQL Server supports two built-in solutions to automatically generate numeric keys:

1. the identity column property
2. the sequence object.

Identity is a standard column property. You can define this property for a column with any numeric type with a scale of zero (no fraction). When defining the property, you can optionally specify a seed (the first value) and an increment (a step value). If you don't provide those, the default is 1 for both. You typically use this property to generate *surrogate keys*, which are keys that are produced by the system and are not derived from the application data.

For example, the following code creates a table called *dbo.T1*:

```
DROP TABLE IF EXISTS dbo.T1;
CREATE TABLE dbo.T1
(
  keycol INT NOT NULL IDENTITY(1, 1) --start(seed) at 1 increment by 1
  CONSTRAINT PK_T1 PRIMARY KEY,
  datacol VARCHAR(10) NOT NULL
  CONSTRAINT CHK_T1_datacol CHECK(datacol LIKE '[ABCDEFGHIJKLMNOPQRSTUVWXYZ]')
);
```

In your *INSERT* statements, you must completely ignore the identity column. For example, the following code inserts three rows into the table, specifying values only for the column *datacol*:

```
INSERT INTO dbo.T1(datacol) VALUES('AAAAA'),('CCCCC'),('BBBBB');
```

When you query the table, naturally you can refer to the identity column by its name (*keycol* in this case). SQL Server also provides a way to refer to the identity column by using the more generic form *\$identity*.

For example, the following query selects the identity column from *T1* by using the generic form:

```
SELECT $identity FROM dbo.T1;
```

keycol
1
2
3

When you insert a new row into the table, SQL Server generates a new identity value based on the current identity value in the table and the increment. If you need to obtain the newly generated identity value—for example, to insert child rows into a referencing table—you query one of two functions, called

1. *@@identity*
2. *SCOPE_IDENTITY*

@@identity function: returns the last identity value generated by the session, regardless of scope (for example, a procedure issuing an *INSERT* statement, and a trigger fired by that statement are in different scopes)

SCOPE_IDENTITY: returns the last identity value generated by the current scope (for example, the same procedure). Except in the rare cases when you don't really care about scope, you should use the *SCOPE_IDENTITY* function.

Remember that both *@@identity* and *SCOPE_IDENTITY* return the last identity value produced by the current session. Neither is affected by inserts issued by other sessions. However, if you want to know the current identity value in a table (the last value produced) regardless of session, you should use the ***IDENT_CURRENT*** function and provide the table name as input.

For example, run the following code from a new session(new query) (not the one from which you ran the previous *INSERT* statements

1

```
Query4.sql - DE...0CTCC3\tench (60))* -> X
SELECT
SCOPE_IDENTITY() AS [SCOPE_IDENTITY],
@@identity AS [@@identity],
IDENT_CURRENT(N'dbo.T1') AS [IDENT_CURRENT];
```

SCOPE_IDENTITY	@@identity	IDENT_CURRENT
NULL	NULL	4

2

```
Query3.sql - DE...0CTCC3\tench (51))* -> X
DECLARE @new_key AS INT;
INSERT INTO dbo.T1(datacol) VALUES('AAAAA');
SET @new_key = SCOPE_IDENTITY();
SELECT @new_key AS new_key
```

new_key
5

3

```
Query4.sql - DE...0CTCC3\tench (60))* -> X
SELECT
SCOPE_IDENTITY() AS [SCOPE_IDENTITY],
@@identity AS [@@identity],
IDENT_CURRENT(N'dbo.T1') AS [IDENT_CURRENT];
```

SCOPE_IDENTITY	@@identity	IDENT_CURRENT
NULL	NULL	5

IDENT_CURRENT: returns the current identity for the table regardless of the session(query)

Both *@@identity* and *SCOPE_IDENTITY* returned *NULL*s because no identity values were created in the session in which this query ran. *IDENT_CURRENT* returned the value 4 because it returns the current identity value in the table, regardless of the session in which it was produced.

SCOPE_IDENTITY and @@IDENTITY return the last identity values that are generated in any table in the current session. However, SCOPE_IDENTITY returns values inserted only within the current scope; @@IDENTITY is not limited to a specific scope.

The change to the current identity value in a table is not undone if the *INSERT* that generated the change fails or the transaction in which the statement runs is rolled back. For example, run the following *INSERT* statement, which conflicts with the *CHECK* constraint defined in the table:

```
INSERT INTO dbo.T1(datacol) VALUES('12345');
```

The insert fails, and you get the following error:

Msg 547, Level 16, State 0, Line 159 The INSERT statement conflicted with the CHECK constraint "CHK_T1_datacol". The conflict occurred in database "TSQLV4", table "dbo.T1", column 'datacol'. The statement has been terminated.

Even though the insert failed, the current identity value in the table changed from 4 to 5, and this change was not undone because of the failure. This means that the next insert will produce the value 6:

```
INSERT INTO dbo.T1(datacol) VALUES('EEEE');
```

```
SELECT * FROM dbo.T1;
```

Notice a gap between the values 4 and 6 in the output:

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB
4	AAAAA
6	EEEEE

One of the shortcomings of the identity property is that you cannot add it to an existing column or remove it from an existing column. If you need to make such a change, it's an expensive and cumbersome offline operation.

With SQL Server, you can specify your own explicit values for the identity column when you insert rows, as long as you enable a session option called *IDENTITY_INSERT* against the table involved.

There's no option you can use to update an identity column, though. For example, the following code demonstrates how **to insert a row into T1 with the explicit value 5 in keycol:**

```
SET IDENTITY_INSERT dbo.T1 ON;
INSERT INTO dbo.T1(keycol, datacol) VALUES(5, 'FFFFF');
SET IDENTITY_INSERT dbo.T1 OFF;
```

Interestingly, when you turn off the *IDENTITY_INSERT* option, SQL Server changes the current identity value in the table **only if the explicit value you provided is greater than the current identity value**. Because the current identity value in the table prior to running the preceding code was 6, and the *INSERT* statement in this code used the lower explicit value 5, the current identity value in the table did not change. So if at this point you query the *IDENT_CURRENT* function for this table, you will get 6 and not 5. This way, the next *INSERT* statement against the table will produce the value 7:

If you need to guarantee uniqueness in an identity column, make sure you also define a primary key or a unique constraint on that column.

Sequence

T-SQL supports the standard sequence object as an alternative key-generating mechanism for identity. The sequence object is more flexible than identity in many ways, making it the preferred choice in many cases.

One of the advantages of the sequence object is that, unlike identity, it's not tied to a particular column in a particular table; rather, it's an independent object in the database. Whenever you need to generate a new value, **you invoke a function against the object and use the returned value wherever you like.** For example, if you have such a use case, you can use one sequence object that will help you maintain keys that will not conflict across multiple tables.

CREATE SEQUENCE

- To create a sequence object, use the *CREATE SEQUENCE* command. The minimum required information is just the sequence name, but note that the defaults for the various properties in such a case might not be what you want. If you don't indicate the data type, SQL Server will use *BIGINT* by default. If you want a different type, indicate *AS <type>*. The type can be any numeric type with a scale of zero. For example, if you need your sequence to be of an *INT* type, indicate *AS INT*.

START WITH <constant>

- The first value returned by the sequence object. The **START** value must be a value less than or equal to the maximum and greater than or equal to the minimum value of the sequence object. The default start value for a new sequence object is the minimum value for an ascending sequence object and the maximum value for a descending sequence object.

INCREMENT BY <constant>

- Value used to increment (or decrement if negative) the value of the sequence object for each call to the **NEXT VALUE FOR** function. If the increment is a negative value, the sequence object is descending; otherwise, it is ascending. The increment cannot be 0. The default increment for a new sequence object is 1.

[MINVALUE <constant> | **NO MINVALUE**] [MAXVALUE <constant> | **NO MAXVALUE**

- Unlike the identity property, the sequence object supports the specification of a minimum value (*MINVALUE <val>*) and a maximum value (*MAXVALUE <val>*) within the type. If you don't indicate what the minimum and maximum values are, the sequence object will assume the minimum and maximum values supported by the type. For example, for an *INT* type, those would be -2,147,483,648 and 2,147,483,647, respectively.

[**CYCLE** | **NO CYCLE**]

- Property that specifies whether the sequence object should restart from the minimum value (or maximum for descending sequence objects) or throw an exception when its minimum or maximum value is exceeded. The default cycle option for new sequence objects is **NO CYCLE**.

[**CACHE** [<constant>] | **NO CACHE**]

- The sequence object also supports a caching option (*CACHE <val>* | *NO CACHE*) that tells SQL Server how often to write the recoverable value to disk. For example, if you specify a cache value of 10,000, SQL Server will write to disk every 10,000 requests, and in between disk writes, it will maintain the current value and how many values are left in memory. If you write less frequently to disk, you'll get better performance when generating a value (on average), but you'll risk losing more values in case of an unexpected termination of the SQL Server process, such as in a power failure. SQL Server has a default cache value of 50, although this number is not officially documented because Microsoft wants to be able to change it.
- You can change any of the sequence properties except the data type with the *ALTER SEQUENCE* command (*MINVAL <val>*, *MAXVAL <val>*, *RESTART WITH <val>*, *INCREMENT BY <val>*, *CYCLE* | *NO CYCLE*, or *CACHE <val>* | *NO CACHE*). For example, suppose you want to prevent the sequence *dbo.SeqOrderIDs* from cycling. You can change the current sequence definition with the following *ALTER SEQUENCE* command:

```
ALTER SEQUENCE dbo.SeqOrderIDs
NO CYCLE;
```

For example, suppose you want to create a sequence that will help you generate order IDs. You want it to be of an *INT* type, have a minimum value of 1 and a maximum value that is the maximum supported by the type, start with 1, increment by 1, and allow cycling. Here's the *CREATE SEQUENCE* command you would use to create such a sequence:

```
CREATE SEQUENCE dbo.SeqOrderIDs AS INT
MINVALUE 1
CYCLE;
```

You had to be explicit about the type, minimum value, and cycling option because they are different than the defaults. You didn't need to indicate the maximum, start with, and increment values because you wanted the defaults.

Notice that, unlike with identity, you didn't need to insert a row into a table in order to generate a new value. Some applications need to generate the new value before using it. With sequences, you can store the result of the function in a variable and use it later in the code. To demonstrate this, first create a table called *T1* with the following code:

```
DROP TABLE IF EXISTS dbo.T1;
CREATE TABLE dbo.T1
(
    keycol INT NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
);
```

The following code generates a new sequence value, stores it in a variable, and then uses the variable in an *INSERT* statement to insert a row into the table:

```
DECLARE @neworderid AS INT = NEXT VALUE FOR dbo.SeqOrderIDs;
INSERT INTO dbo.T1(keycol, datacol) VALUES(@neworderid, 'a');

SELECT * FROM dbo.T1;
```

Now if there is an error the identity is not incremented

DELETING DATA

T-SQL provides two statements for deleting rows from a table:

- *DELETE*
- *TRUNCATE*

The ***DELETE*** statement

The *DELETE* statement is a standard statement used to delete data from a table based on an optional filter predicate.

The standard statement has only two clauses

- *FROM* clause, in which you specify the target table name
- *WHERE* clause, in which you specify a predicate.

Only the subset of rows for which the predicate evaluates to *TRUE* will be deleted.

For example, the following statement deletes, from the *dbo.Orders* table, all orders that were placed prior to 2015:

```
DELETE FROM dbo.Orders  
WHERE orderdate < '20150101';
```

Note that you can suppress returning the message that indicates the number of rows affected by turning on the session option *NOCOUNT*. As you probably noticed, this option is off by default.

The ***TRUNCATE*** statement

The standard *TRUNCATE* statement deletes all rows from a table. Unlike the *DELETE* statement, *TRUNCATE* has no filter. For example, to delete all rows from a table called *dbo.T1*, you run the following code:

```
TRUNCATE TABLE dbo.T1;
```

The advantage that *TRUNCATE* has over *DELETE* is that the former is minimally logged, whereas the latter is fully logged, resulting in significant performance differences. For example, if you use the *TRUNCATE* statement to delete all rows from a table with millions of rows, the operation will finish in a matter of seconds. If you use the *DELETE* statement, the operation can take many minutes. Note that I said *TRUNCATE* is *minimally* logged, as opposed to not being logged at all. SQL Server records which blocks of data were deallocated by the operation so that it can reclaim those in case the transaction needs to be undone. Both *DELETE* and *TRUNCATE* are transactional.

TRUNCATE VS DELETE

TRUNCATE

1. Minimally logged (Faster)
2. Truncate resets the identity value to the original seed
3. No filters
4. The *TRUNCATE* statement is not allowed when the target table is referenced by a foreign-key constraint, even if the referencing table is empty and even if the foreign key is disabled. The only way to allow a *TRUNCATE* statement is to drop all foreign keys referencing the table with the *ALTER TABLE DROP CONSTRAINT* command. You can then re-create the foreign keys after truncating the table with the *ALTER TABLE ADD CONSTRAINT* command.

DELETE

1. Fully logged (slower)
2. Delete does not reset the identity value to the original seed even without a filter
3. The standard statement has only two clauses
 - *FROM* clause, in which you specify the target table name
 - *WHERE* clause, in which you specify a predicate.

Only the subset of rows for which the predicate evaluates to *TRUE* will be deleted.

Accidents such as truncating or dropping the incorrect table can happen. For example, let's say you have connections open against both the production and the development environments, and you submit your code in the wrong connection. Both the *TRUNCATE* and *DROP* statements are so fast that the transaction is committed before you realize your mistake. To prevent such accidents, you can protect a production table by simply creating a dummy table with a foreign key pointing to that table. You can even disable the foreign key so that it won't have any impact on performance. As I mentioned earlier, even when disabled, this foreign key prevents you from truncating or dropping the referenced table.

DELETE based on a join

T-SQL supports a nonstandard *DELETE* syntax based on joins. The join serves a filtering purpose and also gives you access to the attributes of the related rows from the joined tables. This means you can delete rows from one table based on a filter against attributes in related rows from another table.

```
DELETE FROM 0
FROM dbo.Orders AS O
INNER JOIN dbo.Customers AS C
```

==

```
DELETE FROM dbo.Orders
FROM dbo.Orders AS O
INNER JOIN dbo.Customers AS C
ON O.custid = C.custid
WHERE C.country = 'USA';
```

```
ON O.custid = C.custid  
WHERE C.country = N'USA';
```

start with the *FROM* clause with the joins, move on to the *WHERE* clause, and finally—instead of specifying a *SELECT* clause—specify a *DELETE* clause with the alias of the side of the join that is supposed to be the target for the deletion.

BETTER WAY OF DOING THE ABOVE CODE(Using Subqueries)

```
DELETE FROM dbo.Orders  
WHERE EXISTS  
(SELECT *  
FROM dbo.Customers AS C  
WHERE Orders.custid = C.custid  
AND C.country = N'USA');
```

Much like in a *SELECT* statement, the first clause that is logically processed in a *DELETE* statement is the *FROM* clause (the second one that appears in this statement). Then the *WHERE* clause is processed, and finally the *DELETE* clause.

Logical Order Of Process

- *FROM*
- *WHERE*
- *DELETE*

UPDATING DATA

T-SQL supports a standard *UPDATE* statement you can use to update rows in a table. T-SQL also supports nonstandard forms of the *UPDATE* statement with joins and with variables.

The *UPDATE* statement

The *UPDATE* statement is a standard statement you can use to update a subset of rows in a table.

To identify the subset of rows you need to update

1. you specify a predicate in a *WHERE* clause.
2. You specify the assignment of values to columns in a *SET* clause, separated by commas.

For example, the following *UPDATE* statement increases the discount of all order details for product 51 by 5 percent:

```
UPDATE dbo.OrderDetails
SET discount = discount + 0.05
WHERE productid = 51;
```

T-SQL supports compound assignment operators:

1. += (plus equal)
2. -= (minus equal)
3. *= (multiplication equal)
4. /= (division equal)
5. %= (modulo equal)
6. and others.

You can use these operators to shorten assignment expressions such as the one in the preceding query. Instead of the expression *discount* = *discount* + 0.05, you can use this expression: *discount* += 0.05

Remember that all expressions that appear in the same logical phase are evaluated as a set, logically at the same point in time. Consider the following *UPDATE* statement:

```
UPDATE dbo.T1
SET col1 = col1 + 10, col2 = col1 + 10;
```

Suppose one row in the table has the value 100 in *col1* prior to the update. Can you determine the values of *col1* and *col2* in that row after the update?

If you do not consider the all-at-once concept, you would think that *col1* will be set to 110 and *col2* to 120, as if the assignments were performed from left to right. However, the assignments take place all at once, meaning that both assignments use the same value of *col1*—the value before the update. The result of this update is that both *col1* and *col2* will end up with the value 110.

With the concept of all-at-once in mind, can you figure out how to write an *UPDATE* statement that swaps the values in the columns *col1* and *col2*? In most programming languages where expressions and assignments are evaluated in some order (typically left to right), you need a temporary variable. However, because in SQL all assignments take place as if they happen at the same point in time, the solution is simple:

```
UPDATE dbo.T1
SET col1 = col2, col2 = col1;
```

***UPDATE* based on a join**

Similar to the *DELETE* statement, the *UPDATE* statement also supports a nonstandard form based on joins. As with *DELETE* statements, the join serves a filtering purpose as well as giving you access to attributes from the joined tables.

The syntax involved

- UPDATE
- FROM
- WHERE

LOGICAL PROCESSING

- FROM
- WHERE
- UPDATE

The *UPDATE* keyword is followed by the alias of the table that is the target of the update (you can't update more than one table in the same statement), followed by the *SET* clause with the column assignments.

increases the discount of all order details of orders placed by customer 1 by 5 percent.

```
UPDATE OD
SET discount += 0.05
FROM dbo.OrderDetails AS OD
INNER JOIN dbo.Orders AS O
ON OD.orderid = O.orderid
WHERE O.custid = 1;
```



```
UPDATE dbo.OrderDetails
SET discount += 0.05
WHERE EXISTS
(SELECT * FROM dbo.Orders AS O
WHERE O.orderid =
OrderDetails.orderid
AND O.custid = 1);
```

The query joins the *OrderDetails* table (aliased as *OD*) with the *Orders* table (aliased as *O*) based on a match between the order detail's order ID and the order's order ID. The query then filters only the rows where the order's customer ID is 1. The query then specifies in the *UPDATE* clause that *OD* (the alias of the *OrderDetails* table) is the target of the update, and it increases the discount by 5 percent. You can also specify the full table name in the *UPDATE* clause if you like.

Assignment *UPDATE*

T-SQL supports a proprietary *UPDATE* syntax that both updates data in a table and assigns values to variables at the same time. This syntax saves you the need to use separate *UPDATE* and *SELECT* statements to achieve the same task.

One of the common cases for which you can use this syntax is in maintaining a custom sequence/ autonumbering mechanism when the identity column property and the sequence object don't work for you. One example is when you need to guarantee that there are no gaps between the values. To achieve this, you keep the last-used value in a table, and whenever you need a new value, you use the special *UPDATE* syntax to both increment the value in the table and assign it to a variable.

Run the following code to first create the *MySequences* table with the column *val*, and then populate it with a single row with the value 0—one less than the first value you want to use:

```
DROP TABLE IF EXISTS dbo.MySequences;
CREATE TABLE dbo.MySequences
(
  id VARCHAR(10) NOT NULL
  CONSTRAINT PK_MySequences PRIMARY KEY(id),
  val INT NOT NULL
);
INSERT INTO dbo.MySequences VALUES('SEQ1', 0);
```

```
DECLARE @nextval AS INT;
UPDATE dbo.MySequences
SET @nextval = val += 1
WHERE id = 'SEQ1';
SELECT @nextval;
```

The code declares a local variable called *@nextval*. Then it uses the special *UPDATE* syntax to increment the column value by 1 and assigns the new value to a variable. The code then presents the value in the variable. First *val* is set to *val* + 1, and then the result (*val* + 1) is set to the variable *@nextval*.

The specialized *UPDATE* syntax is run as a transaction, and it's more efficient than using separate *UPDATE* and *SELECT* statements because it accesses the data only once. Note that variable assignment isn't transactional, though.

MERGING DATA

T-SQL supports a statement called *MERGE* you can use to merge data from a source into a target, applying different actions (*INSERT*, *UPDATE*, and *DELETE*) based on conditional logic.

```
SELECT * FROM dbo.Customers;
```

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	cust 2	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	cust 5	(555) 555-5555	address 5

```
SELECT * FROM dbo.CustomersStage;
```

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Suppose you need to merge the contents of the *CustomersStage* table (the source) into the *Customers* table (the target). More specifically, you need to add customers that do not exist and update the customers that do exist.

You specify the target table name in the *MERGE* clause and the source table name in the *USING* clause. You define a merge condition by specifying a predicate in the *ON* clause. The merge condition defines which rows in the source table have matches in the target and which don't. You define the action to take when a match is found in a clause called *WHEN MATCHED THEN*, and the action to take when a match is not found in the *WHEN NOT MATCHED THEN* clause.

Here's the first example for the *MERGE* statement. It adds nonexistent customers and updates existing ones:

```
MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
ON TGT.custid = SRC.custid
WHEN MATCHED THEN
UPDATE SET
TGT.companyname = SRC.companyname,
TGT.phone = SRC.phone,
TGT.address = SRC.address
WHEN NOT MATCHED THEN
INSERT (custid, companyname, phone, address)
VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);
```

This *MERGE* statement defines the *Customers* table as the target (in the *MERGE* clause) and the *CustomersStage* table as the source (in the *USING* clause). Notice that you can assign aliases to the target and source tables for brevity (TGT and SRC in this case). The predicate *TGT.custid = SRC.custid* is used to define what is considered a match and what is considered a nonmatch. In this case, if a customer ID that exists in the source also exists in the target, that's a match. If a customer ID

```
SELECT * FROM dbo.Customers;
```

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	cust 2	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	cust 5	(555) 555-5555	address 5

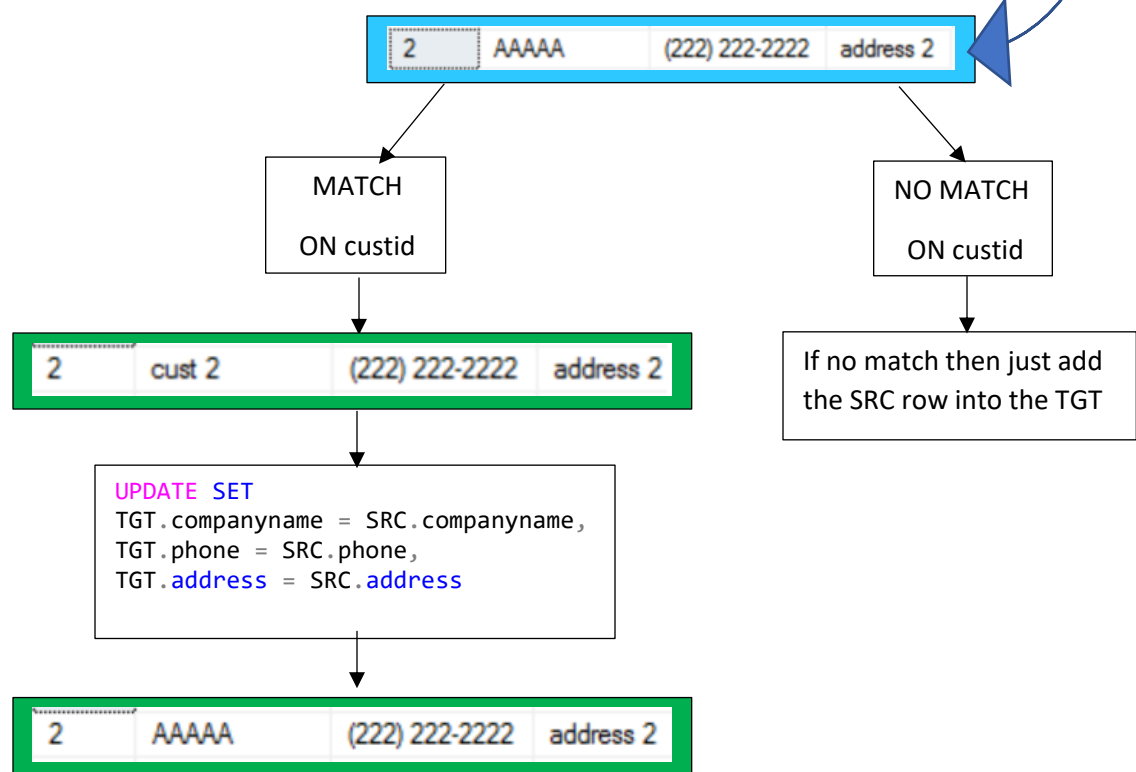
TGT

```
SELECT * FROM dbo.CustomersStage;
```

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

SRC

dbo.Customers AS TGT
dbo.CustomersStage AS SRC



FINAL OUTPUT

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

NOT MATCHED BY SOURCE

```
MERGE dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
ON TGT.custid = SRC.custid
WHEN MATCHED THEN
UPDATE SET
TGT.companyname = SRC.companyname,
TGT.phone = SRC.phone,
TGT.address = SRC.address
WHEN NOT MATCHED THEN
INSERT (custid, companyname, phone, address)
VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
WHEN NOT MATCHED BY SOURCE THEN
DELETE;
```

Deletes everything in the **target** that is not in the **source**

```
SELECT * FROM dbo.Customers;
```

```
SELECT * FROM dbo.CustomersStage;
```

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	cust 2	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	cust 5	(555) 555-5555	address 5

Delete

TGT

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

SRC

FINAL OUTPUT

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Going back to the first *MERGE* example, which updates existing customers and adds nonexistent ones, you can see that it doesn't check whether column values are actually different before applying an update. This means that a customer row is modified even when the source and target rows are identical. If you want to apply the update only if at least one column value is different, there is a way to achieve this.

SOLUTION

The *MERGE* statement supports adding a predicate to the different action clauses by using the *AND* option; the action will take place only if the additional predicate evaluates to *TRUE*. In this case, you need to add a predicate under the *WHEN MATCHED AND* clause that checks that at least one of the column values is different to justify the *UPDATE* action. The complete *MERGE* statement looks like this:

```
USING dbo.CustomersStage AS SRC
ON TGT.custid = SRC.custid
WHEN MATCHED AND
( TGT.companyname <> SRC.companyname
OR TGT.phone <> SRC.phone
OR TGT.address <> SRC.address) THEN
UPDATE SET
TGT.companyname = SRC.companyname,
TGT.phone = SRC.phone,
TGT.address = SRC.address
WHEN NOT MATCHED THEN
INSERT (custid, companyname, phone, address)
VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);
```

<> SAME AS !=

```
( TGT.companyname != SRC.companyname
OR TGT.phone != SRC.phone
OR TGT.address != SRC.address)
```

Modifying data through table expressions

T-SQL doesn't limit the actions against table expressions to *SELECT* only; it also allows other DML statements (*INSERT*, *UPDATE*, *DELETE*, and *MERGE*) against those. Think about it: as explained in Chapter 5, a table expression doesn't really contain data—it's a reflection of data in underlying tables. With this in mind, think of a modification against a table expression as modifying the data in the underlying tables through the table expression. Just as with a *SELECT* statement against a table expression, a modification statement against a table expression also gets expanded, so in practice the activity is done against the underlying tables.

Modifying data through table expressions has a few restrictions:

- If the query defining the table expression joins tables, you're allowed to affect only one of the sides of the join, not both, in the same modification statement.
- You cannot update a column that is a result of a calculation; SQL Server doesn't try to reverse-engineer the values.
- *INSERT* statements must specify values for any columns in the underlying table that do not get their values implicitly. Examples for cases where a column can get a value implicitly include a column that allows *NULLs*, has a default value, has an identity property, or is typed as *ROWVERSION*.

Suppose, for troubleshooting purposes, you first want to see which rows would be modified by this statement without actually modifying them. One option is to revise the code to a *SELECT* statement, and after troubleshooting the code, change it back to an *UPDATE* statement. But instead of needing to make such revisions, you define a table expression based on a *SELECT* statement with the join query and issue an *UPDATE* statement against the table expression.

The following example uses a CTE:

```
WITH C AS
(
    SELECT custid, OD.orderid,
    productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
    INNER JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
    WHERE O.custid = 1
)
UPDATE C
SET discount = newdiscount;
```

custid	orderid	productid	discount	newdiscount
1	10643	28	0.450	0.500
1	10643	39	0.450	0.500
1	10643	46	0.450	0.500
1	10692	63	0.200	0.250
1	10702	3	0.200	0.250
1	10702	76	0.200	0.250
1	10835	59	0.200	0.250
1	10835	77	0.400	0.450
1	10952	6	0.250	0.300
1	10952	28	0.200	0.250
1	11011	58	0.250	0.300
1	11011	71	0.200	0.250

The following example uses a Derived Table:

```
UPDATE D
SET discount = newdiscount
FROM ( SELECT custid, OD.orderid,
    productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
    INNER JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
    WHERE O.custid = 1 ) AS D;
```

