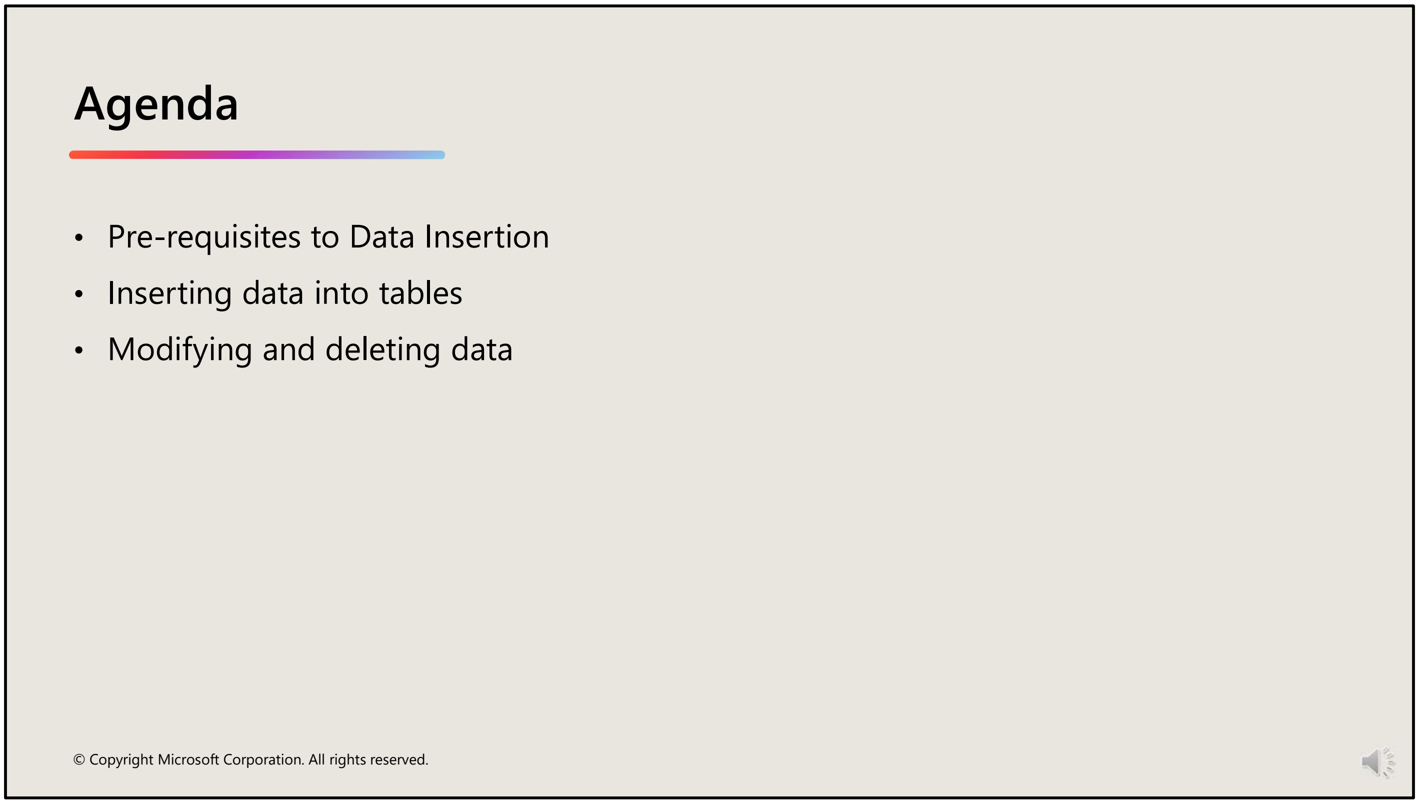
Microsoft

7: Managing Data (Insert/Update/Delete)



Agenda

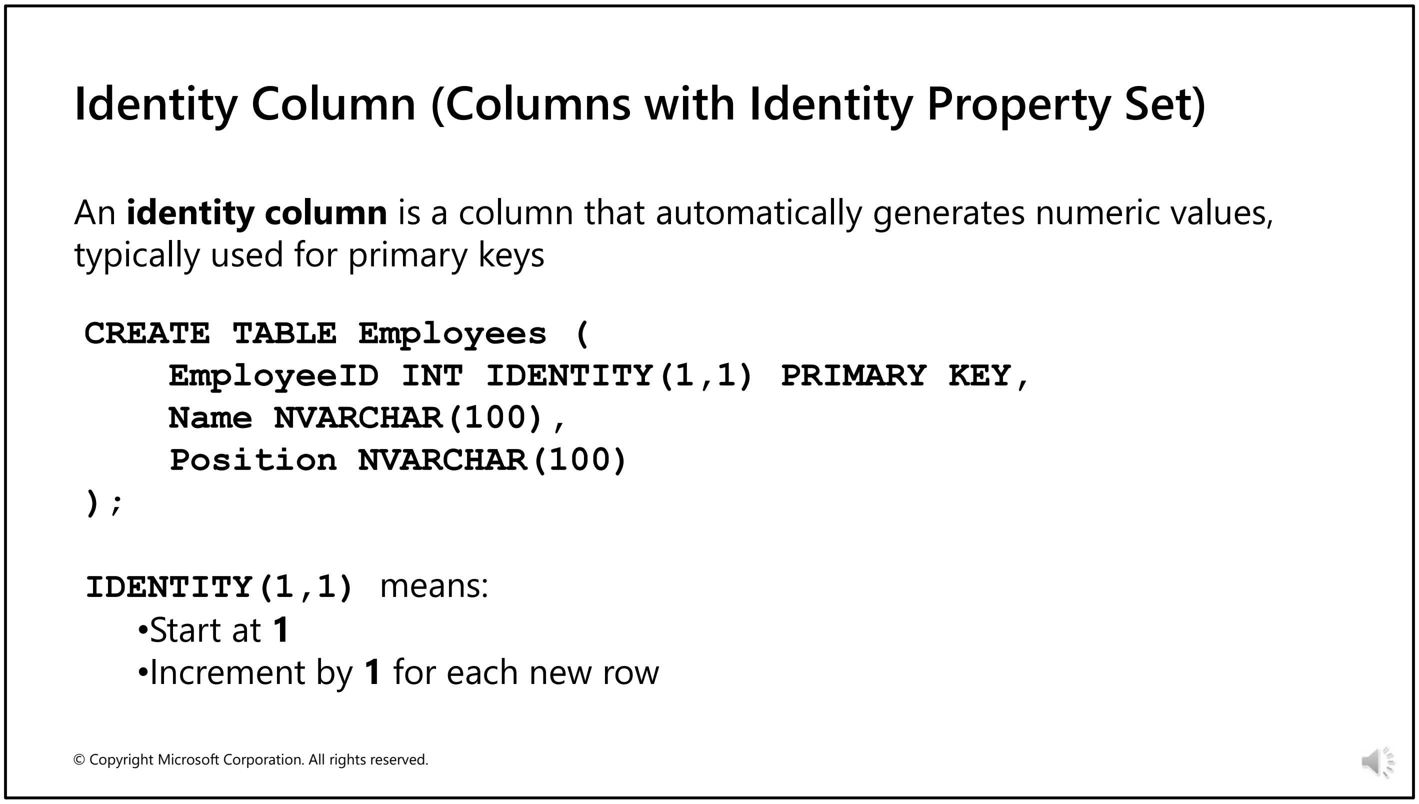
- Pre-requisites to Data Insertion
- Inserting data into tables
- Modifying and deleting data

1: Pre-requisites to Insert Data

In many cases, data analysts and business users simply need to retrieve data from a database for reporting or analysis. However, when developing an application, or even during some complex analysis, you may need to insert, modify, or delete data.

In this module, you'll learn how to:

- Insert data into a table
- Generate automatic values
- Update data in a table
- Delete data from a table
- Merge data based on multiple tables



Identity Column (Columns with Identity Property Set)

An **identity column** is a column that automatically generates numeric values, typically used for primary keys

```
CREATE TABLE Employees (
    EmployeeID INT IDENTITY(1,1) PRIMARY KEY,
    Name NVARCHAR(100),
    Position NVARCHAR(100)
);
```

`IDENTITY(1,1)` means:

- Start at **1**
- Increment by **1** for each new row

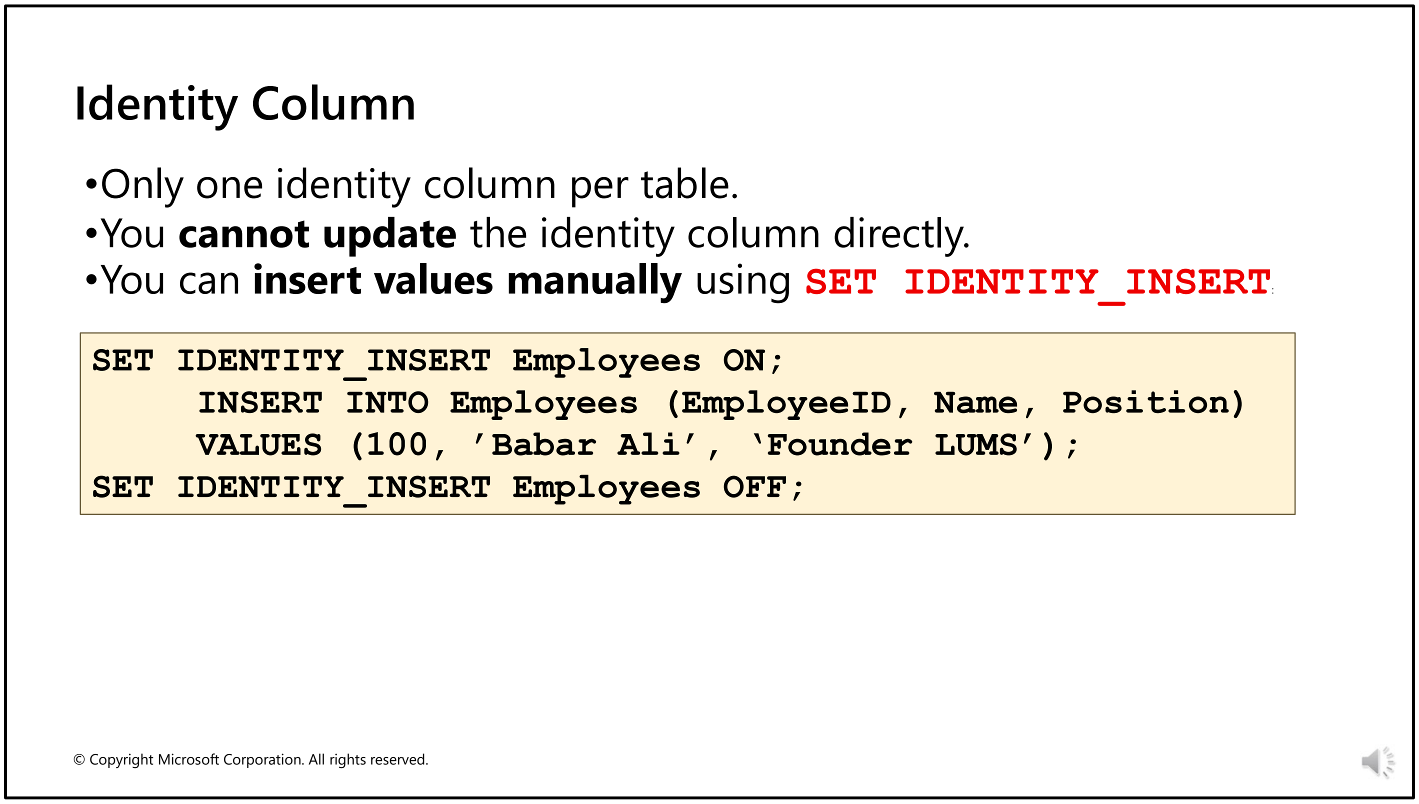
Identity columns can be used for generating key values. The identity property on a column guarantees the following conditions:

- Each new value is generated based on the current seed and increment.
- Each new value for a particular transaction is different from other concurrent transactions on the table.

The identity property on a column doesn't guarantee the following conditions:

- Uniqueness of the value - Uniqueness must be enforced by using a PRIMARY KEY or UNIQUE constraint or UNIQUE index.

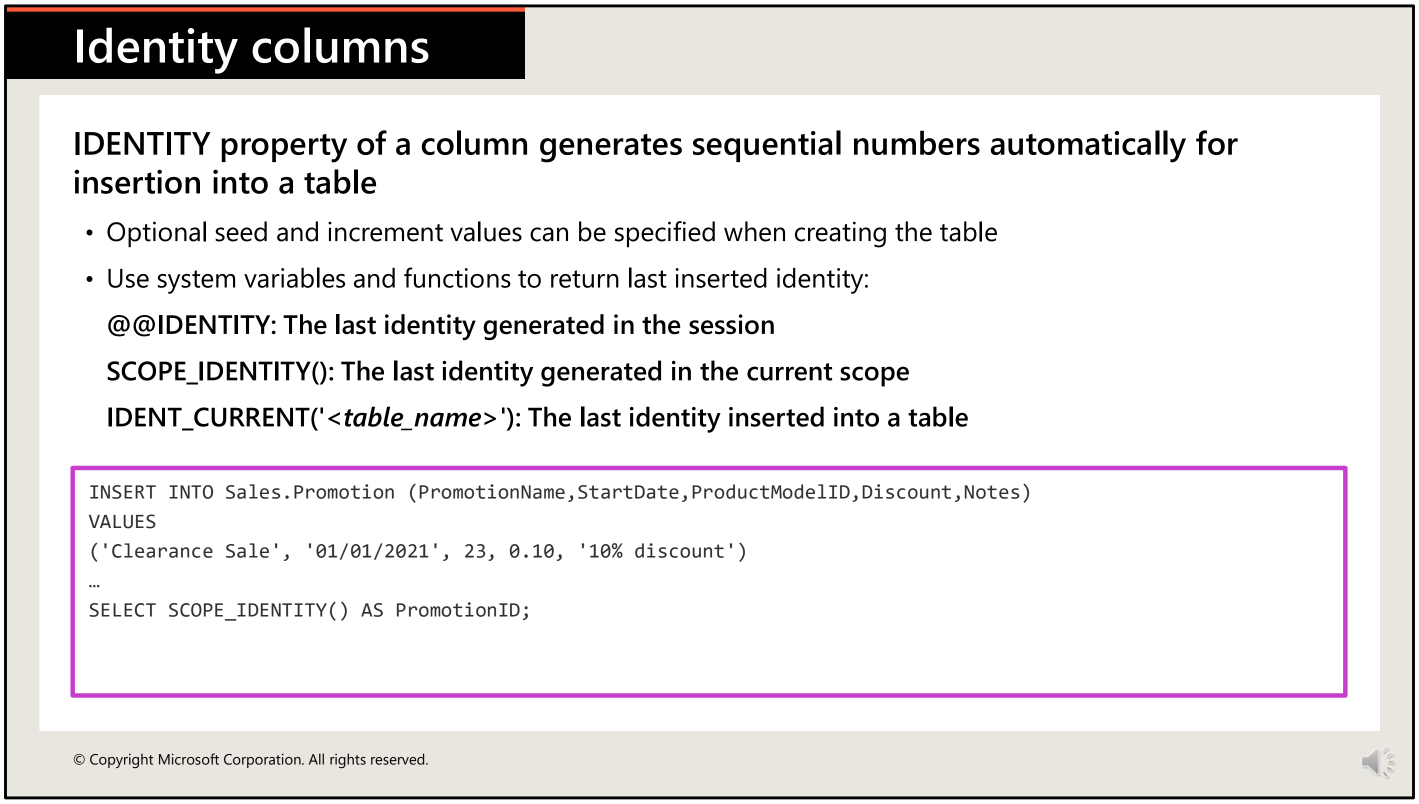
Only one identity column can be created per table.



Identity Column

- Only one identity column per table.
- You **cannot update** the identity column directly.
- You can **insert values manually** using `SET IDENTITY_INSERT`

```
SET IDENTITY_INSERT Employees ON;
     INSERT INTO Employees (EmployeeID, Name, Position)
     VALUES (100, ’Babar Ali’, ‘Founder LUMS’);
SET IDENTITY_INSERT Employees OFF;
```



Identity columns

IDENTITY property of a column generates sequential numbers automatically for insertion into a table

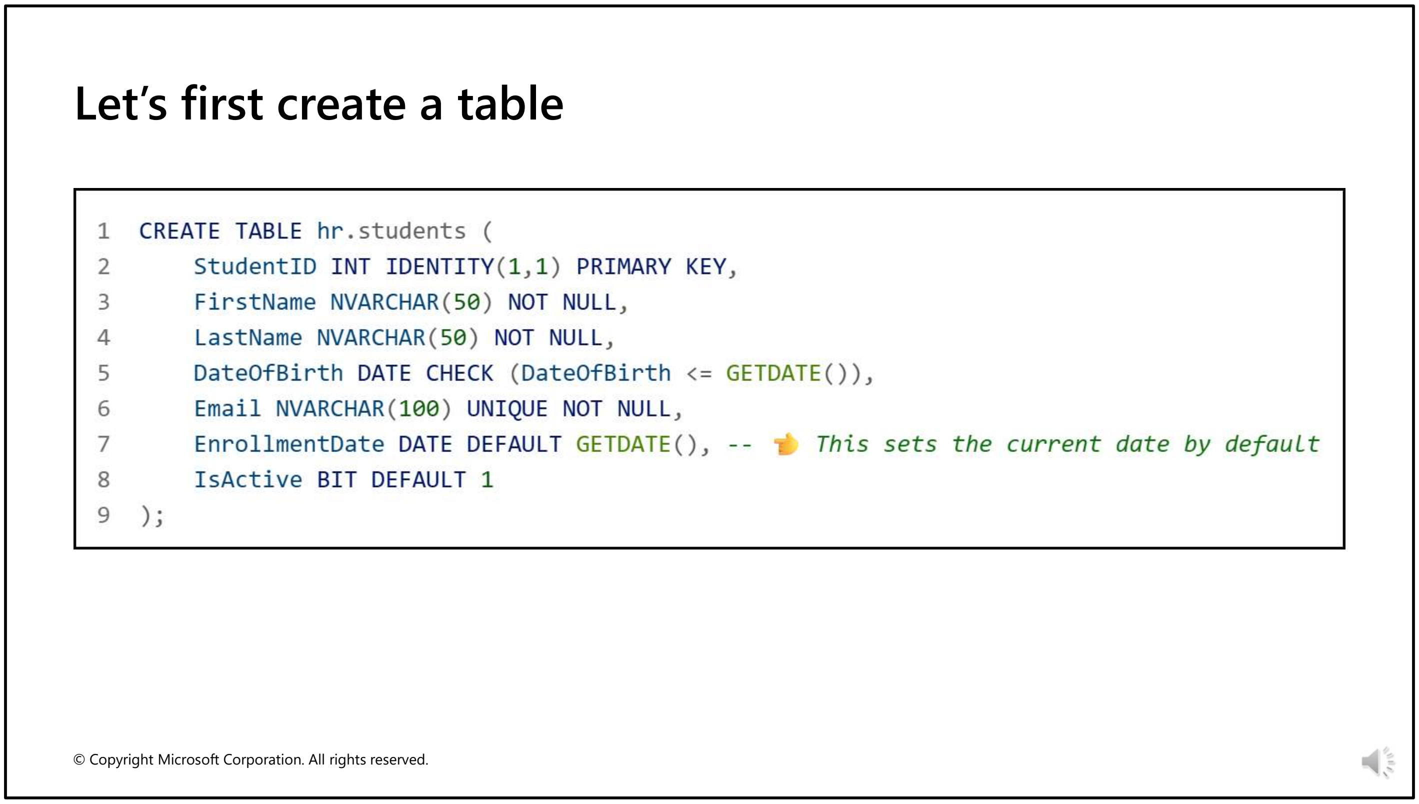
- Optional seed and increment values can be specified when creating the table
- Use system variables and functions to return last inserted identity:

  @@IDENTITY: The last identity generated in the session

  SCOPE_IDENTITY(): The last identity generated in the current scope

  IDENT_CURRENT('<*table_name*>'): The last identity inserted into a table

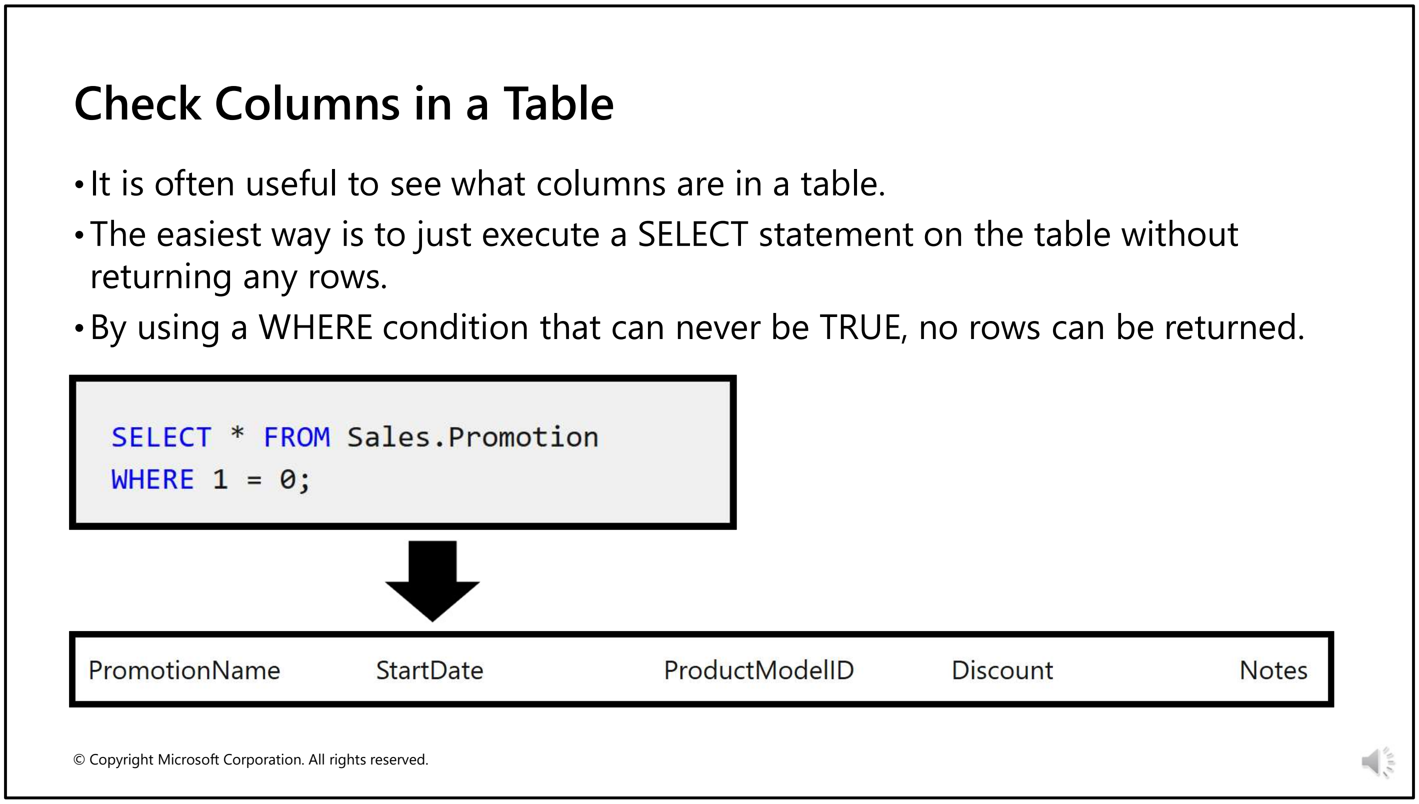
```
INSERT INTO Sales.Promotion (PromotionName,StartDate,ProductModelID,Discount,Notes)
VALUES
('Clearance Sale', '01/01/2021', 23, 0.10, '10% discount')
…
SELECT SCOPE_IDENTITY() AS PromotionID;
```



Let’s first create a table

```sql
CREATE TABLE hr.students (
    StudentID INT IDENTITY(1,1) PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DateOfBirth DATE CHECK (DateOfBirth <= GETDATE()),
    Email NVARCHAR(100) UNIQUE NOT NULL,
    EnrollmentDate DATE DEFAULT GETDATE(), -- 👉 This sets the current date by default
    IsActive BIT DEFAULT 1
);
```

```sql
CREATE TABLE hr.students (
    StudentID INT IDENTITY(1,1) PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DateOfBirth DATE CHECK (DateOfBirth <= GETDATE()),
    Email NVARCHAR(100) UNIQUE NOT NULL,
    EnrollmentDate DATE DEFAULT GETDATE(),
    IsActive BIT DEFAULT 1
);
```

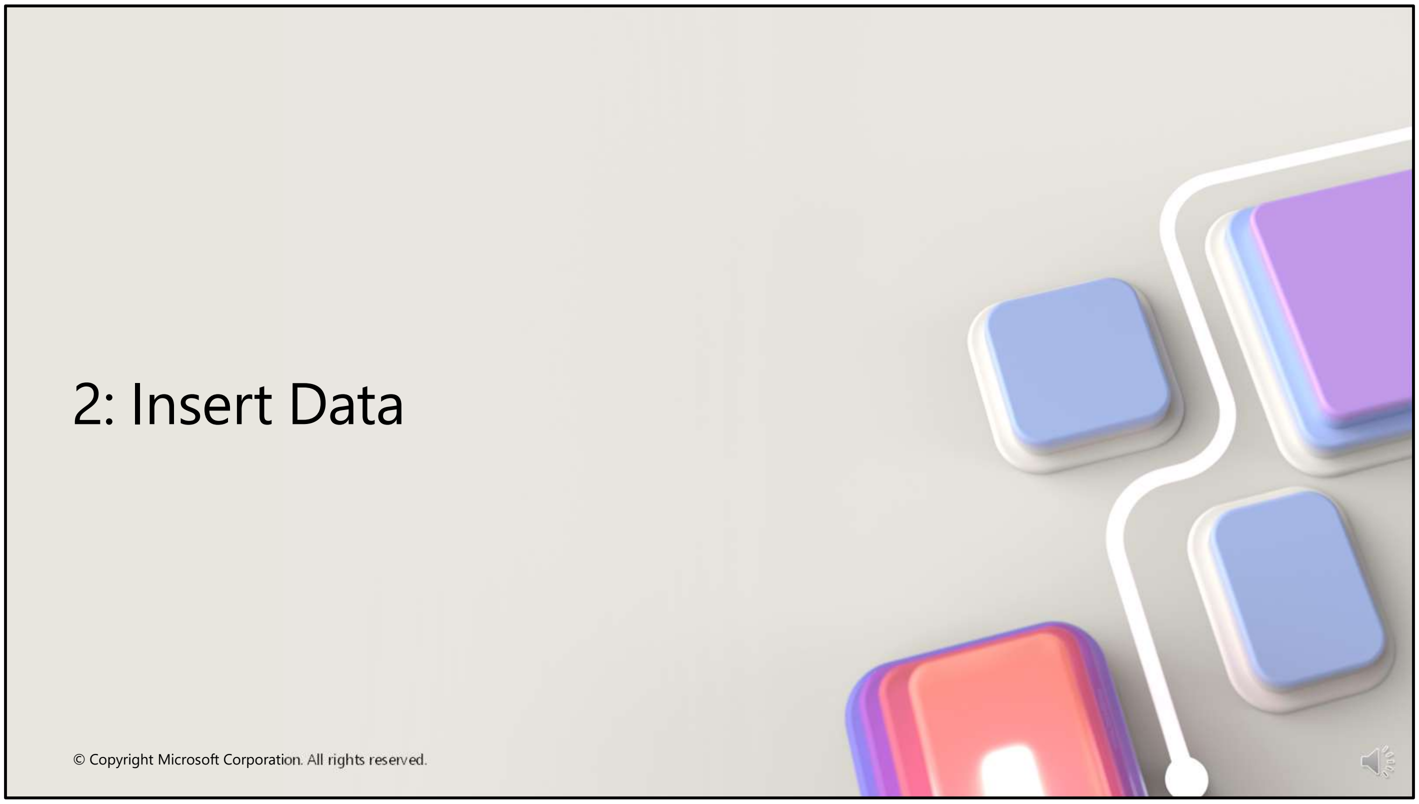


Check Columns in a Table

- It is often useful to see what columns are in a table.
- The easiest way is to just execute a SELECT statement on the table without returning any rows.
- By using a WHERE condition that can never be TRUE, no rows can be returned.

```sql
SELECT * FROM Sales.Promotion
WHERE 1 = 0;
```

PromotionName	StartDate	ProductModelID	Discount	Notes

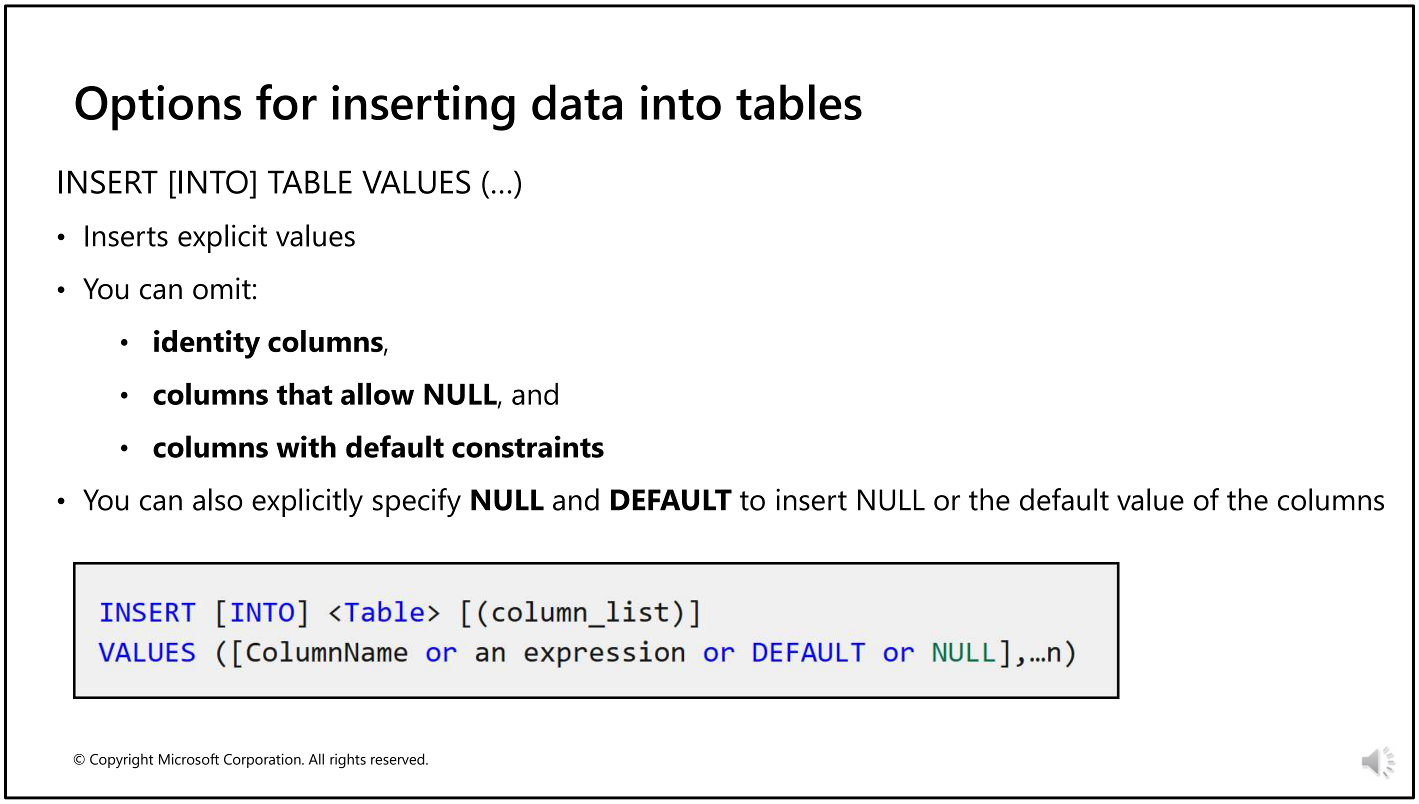


2: Insert Data

In many cases, data analysts and business users simply need to retrieve data from a database for reporting or analysis. However, when developing an application, or even during some complex analysis, you may need to insert, modify, or delete data.

In this module, you'll learn how to:

- Insert data into a table
- Generate automatic values
- Update data in a table
- Delete data from a table
- Merge data based on multiple tables



Options for inserting data into tables

INSERT [INTO] TABLE VALUES (…)

- Inserts explicit values
- You can omit:
  - **identity columns**,
  - **columns that allow NULL**, and
  - **columns with default constraints**
- You can also explicitly specify **NULL** and **DEFAULT** to insert NULL or the default value of the columns

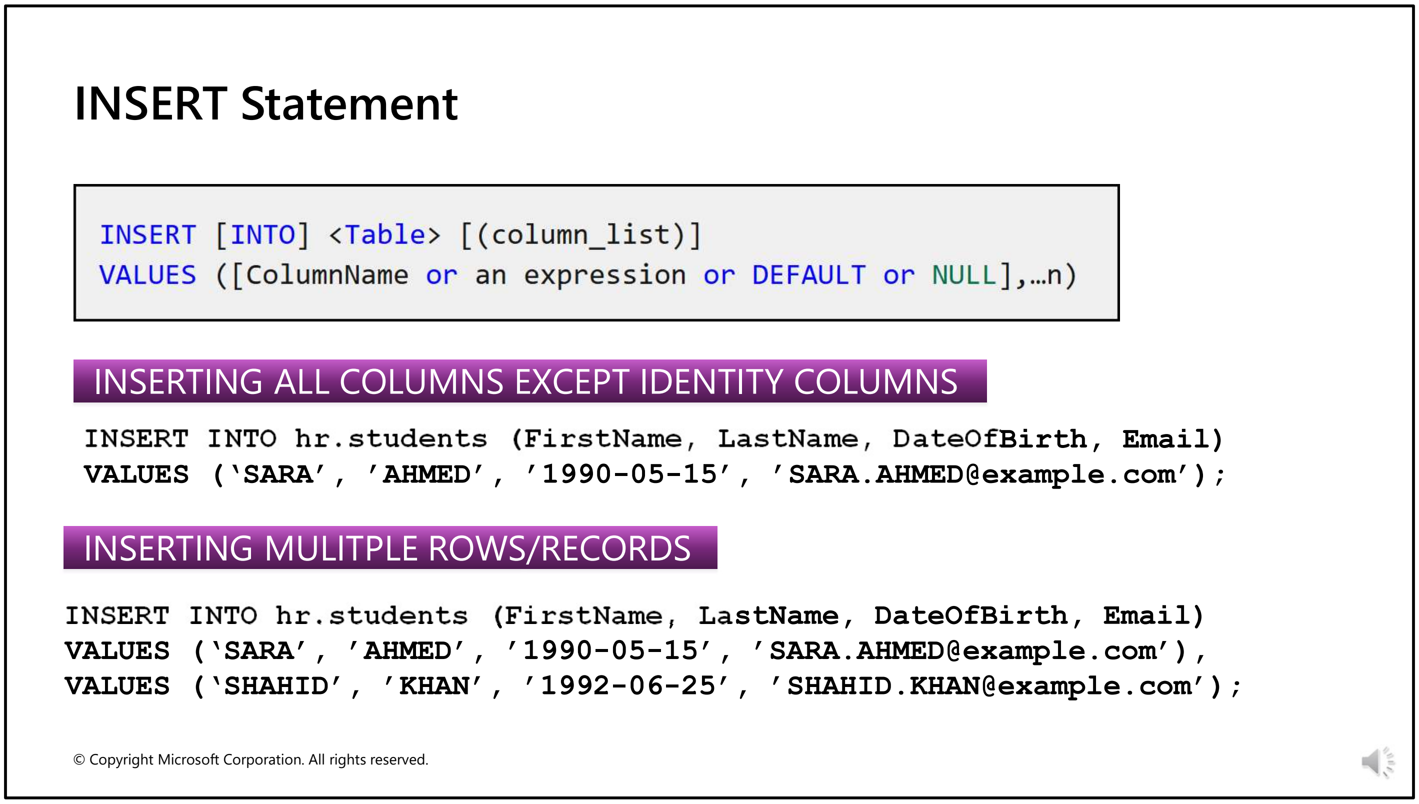
```
INSERT [INTO] <Table> [(column_list)]
VALUES ([ColumnName or an expression or DEFAULT or NULL],…n)
```

https://learn.microsoft.com/en-us/training/modules/modify-data-with-transact-sql/2-insert-data

With this form of the INSERT statement, called INSERT VALUES, you can specify the columns that will have values placed in them and the order in which the data will be presented for each row inserted into the table. The column_list is optional but recommended. Without the column_list, the INSERT statement will expect a value for every column in the table in the order in which the columns were defined. You can also provide the values for those columns as a comma-separated list.

When listing values, the keyword DEFAULT means a predefined value, that was specified when the table was created, will be used. There are three ways a default can be determined:

- If a column has been defined to have an automatically generated value, that value will be used. Autogenerated values will be discussed later in this module.
- When a table is created, a default value can be supplied for a column, and that value will be used if DEFAULT is specified.
- If a column has been defined to allow NULL values, and the column isn't an autogenerated column and doesn't have a default defined, NULL will be inserted as a DEFAULT.



INSERT Statement

```
INSERT [INTO] <Table> [(column_list)]
VALUES ([ColumnName or an expression or DEFAULT or NULL],…n)
```

INSERTING ALL COLUMNS EXCEPT IDENTITY COLUMNS

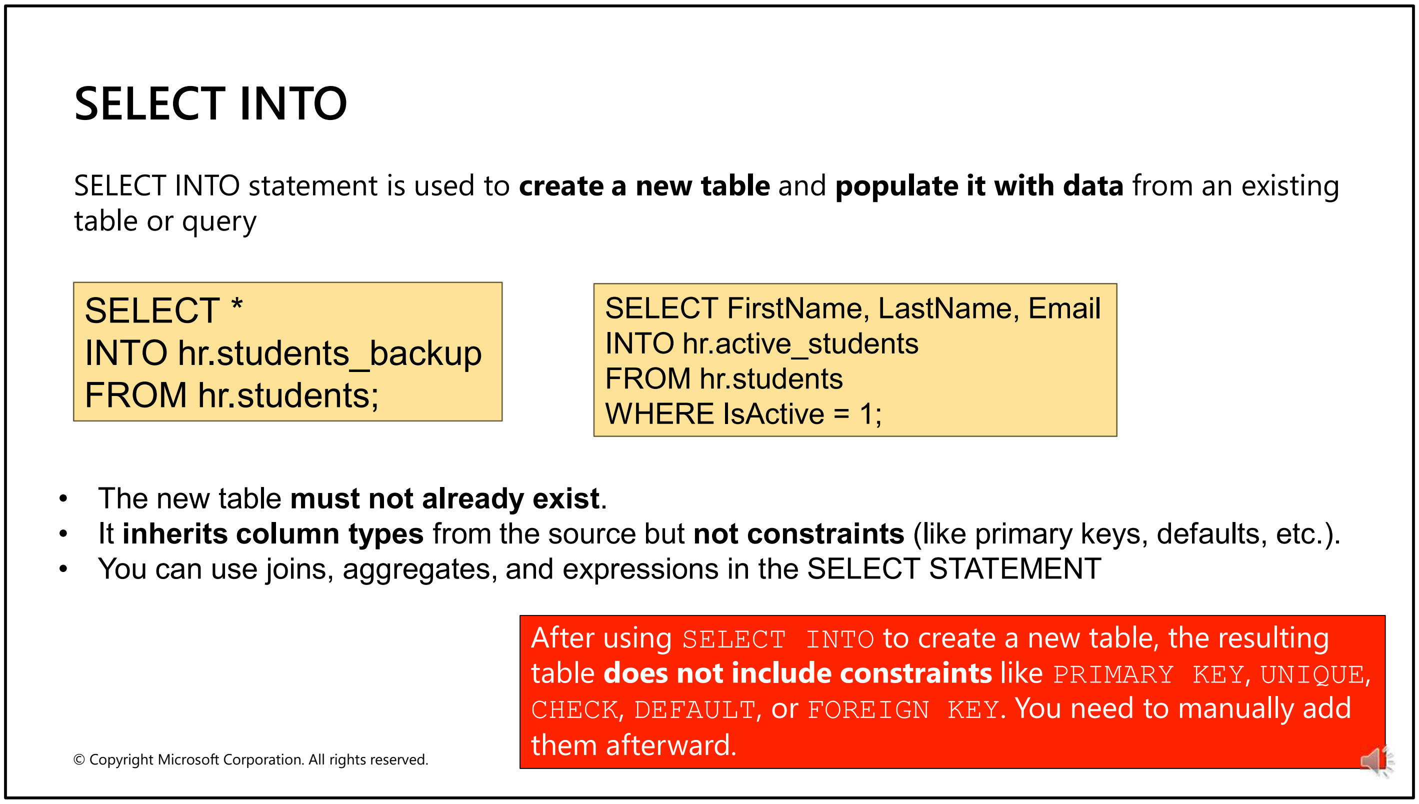
```
INSERT INTO hr.students (FirstName, LastName, DateOfBirth, Email)
VALUES (‘SARA’, ’AHMED’, ’1990-05-15’, ’SARA.AHMED@example.com’);
```

INSERTING MULITPLE ROWS/RECORDS

```
INSERT INTO hr.students (FirstName, LastName, DateOfBirth, Email)
VALUES (‘SARA’, ’AHMED’, ’1990-05-15’, ’SARA.AHMED@example.com’),
VALUES (‘SHAHID’, ’KHAN’, ’1992-06-25’, ’SHAHID.KHAN@example.com’);
```

https://learn.microsoft.com/en-us/training/modules/modify-data-with-transact-sql/2-insert-data

```
INSERT INTO sales.promotions ( promotion_name, discount, start_date, expired_date )
VALUES ( '2018 Summer Promotion', 0.15, '20180601', '20180831' );
```



SELECT INTO

SELECT INTO statement is used to **create a new table** and **populate it with data** from an existing table or query

```
SELECT *
INTO hr.students_backup
FROM hr.students;
```

```
SELECT FirstName, LastName, Email
INTO hr.active_students
FROM hr.students
WHERE IsActive = 1;
```

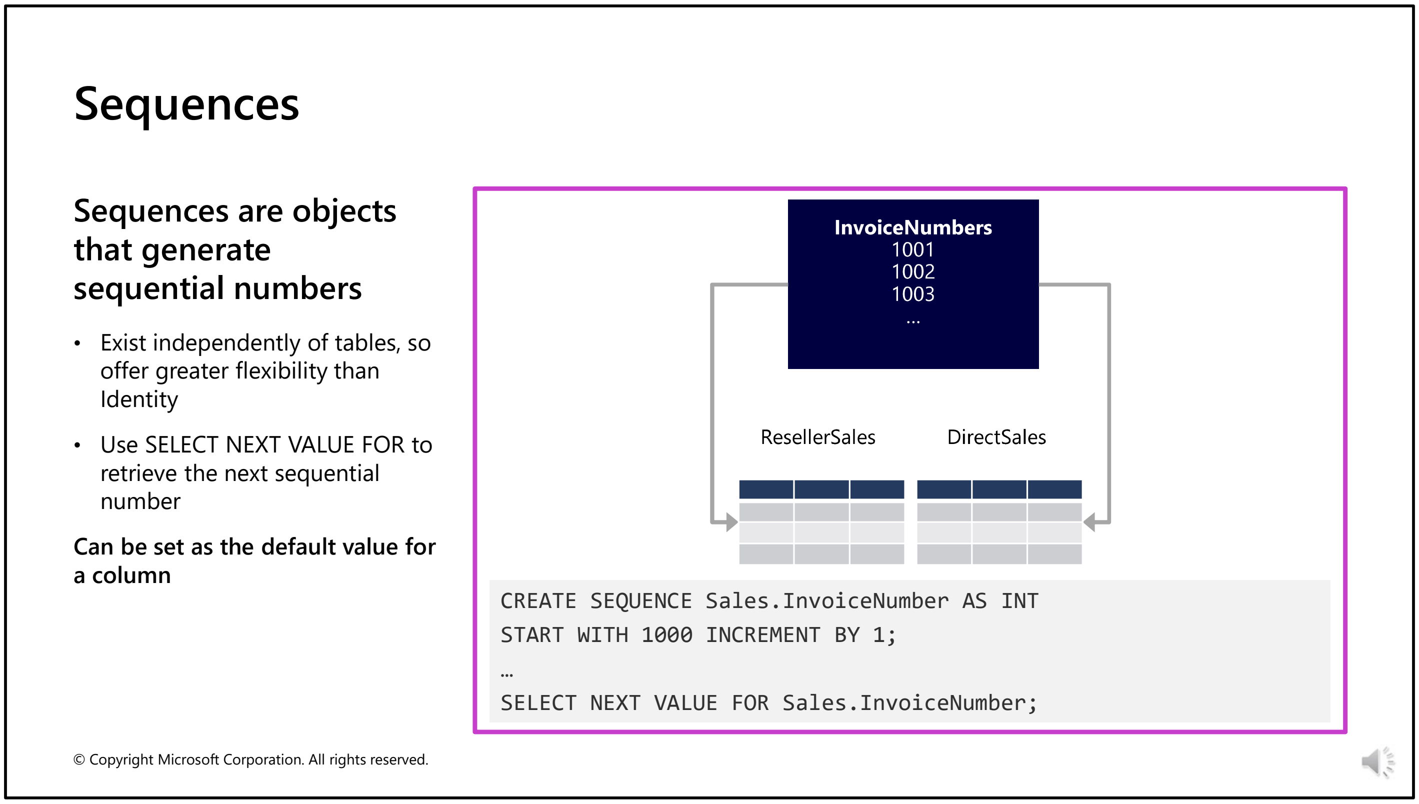
- The new table **must not already exist**.
- It **inherits column types** from the source but **not constraints** (like primary keys, defaults, etc.).
- You can use joins, aggregates, and expressions in the SELECT STATEMENT

After using `SELECT INTO` to create a new table, the resulting table **does not include constraints** like `PRIMARY KEY`, `UNIQUE`, `CHECK`, `DEFAULT`, or `FOREIGN KEY`. You need to manually add them afterward.

SELECT INTO: Adding Constraints to Table

```
ALTER TABLE hr.students_backup
ADD CONSTRAINT PK_students_backup PRIMARY KEY (StudentID);
```

```
ALTER TABLE hr.students_backup
ADD CONSTRAINT UQ_students_backup_Email UNIQUE (Email);
```

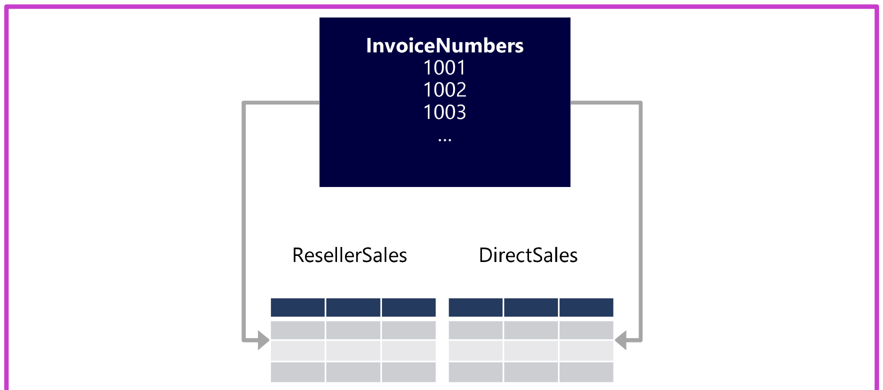


Sequences

Sequences are objects that generate sequential numbers

- Exist independently of tables, so offer greater flexibility than Identity
- Use SELECT NEXT VALUE FOR to retrieve the next sequential number

Can be set as the default value for a column



```
CREATE SEQUENCE Sales.InvoiceNumber AS INT
START WITH 1000 INCREMENT BY 1;
…
SELECT NEXT VALUE FOR Sales.InvoiceNumber;
```

Sequences, unlike identity columns, are not associated with tables. An application refers to a sequence object to receive its next value.

The relationship between sequences and tables is controlled by the application. User applications can reference a sequence object and coordinate the values keys across multiple rows and tables.

A sequence is created independently of the tables by using the CREATE SEQUENCE statement. Options enable you to control the increment, maximum and minimum values, starting point, automatic restarting capability, and caching to improve performance.

Unlike identity column values, which are generated when rows are inserted, an application can obtain the next sequence number before inserting the row by calling the NEXT VALUE FOR function. The sequence number is allocated when NEXT VALUE FOR is called even if the number is never inserted into a table. The NEXT VALUE FOR function can be used as the default value for a column in a table definition.

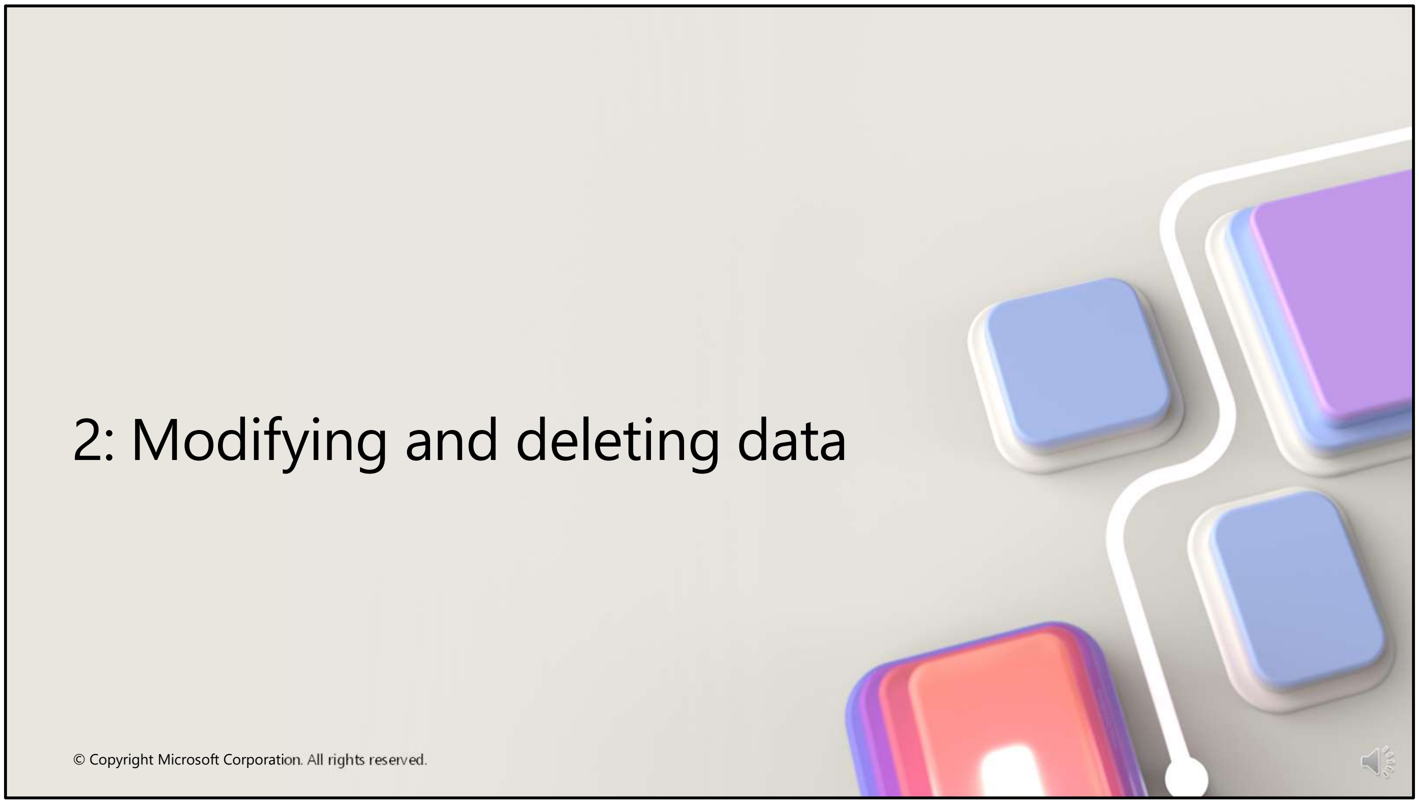
```
CREATE TABLE hr.students (
    StudentID INT DEFAULT NEXT VALUE FOR StudentSeq PRIMARY KEY,
.
.
.
)
```

Use sequences instead of identity columns in the following scenarios:

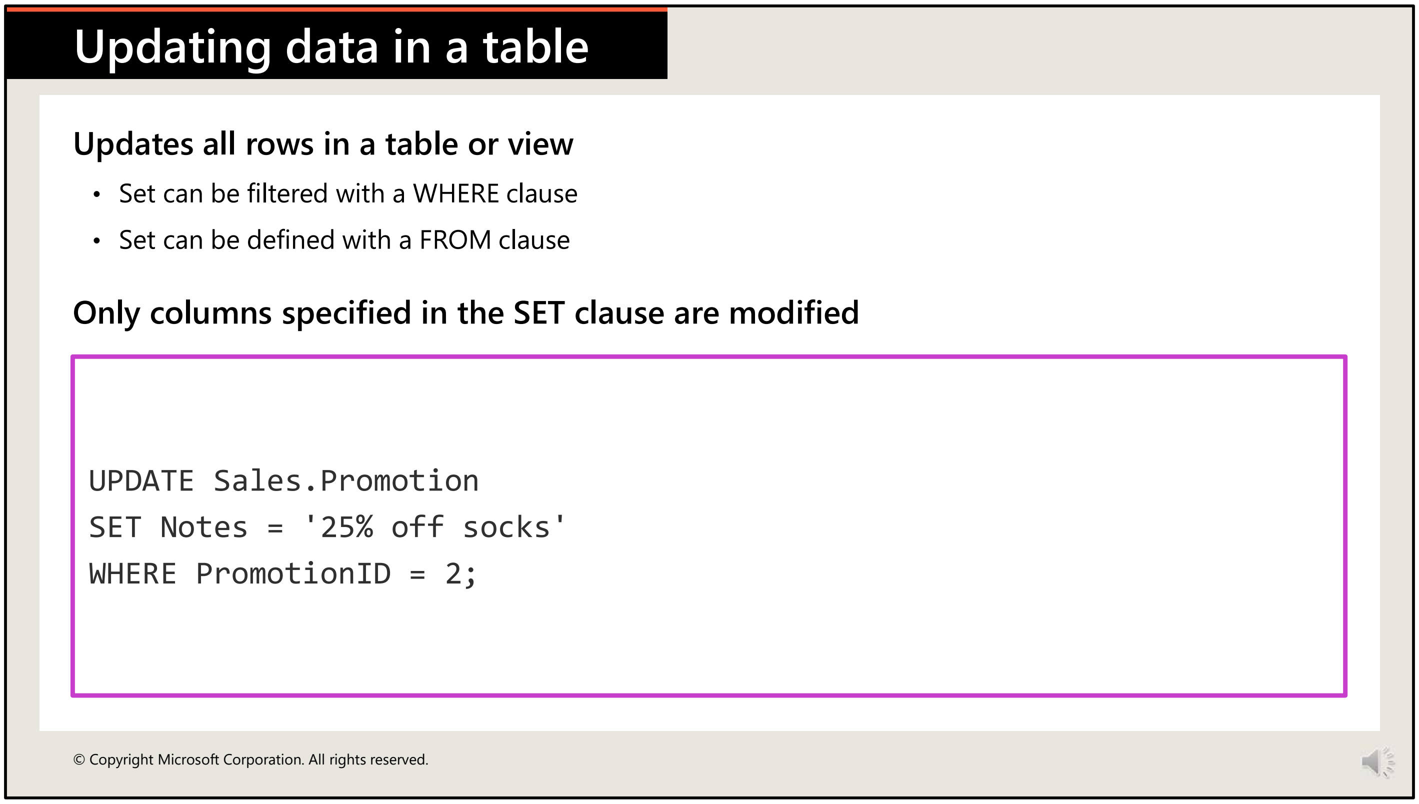
- The application requires a number before the insert into the table is made.
- The application requires sharing a single series of numbers between multiple tables or multiple columns within a table.
- The application must restart the number series when a specified number is reached. For example, after

assigning values 1 through 10, the application starts assigning values 1 through 10 again.

- The application requires sequence values to be sorted by another field. The NEXT VALUE FOR function can apply the OVER clause to the function call. The OVER clause guarantees that the values returned are generated in the order of the OVER clause's ORDER BY clause.
- An application requires multiple numbers to be assigned at the same time. For example, an application needs to reserve five sequential numbers. Requesting identity values could result in gaps in the series if other processes were simultaneously issued numbers. Calling sp_sequence_get_range can retrieve several numbers in the sequence at once.



2: Modifying and deleting data



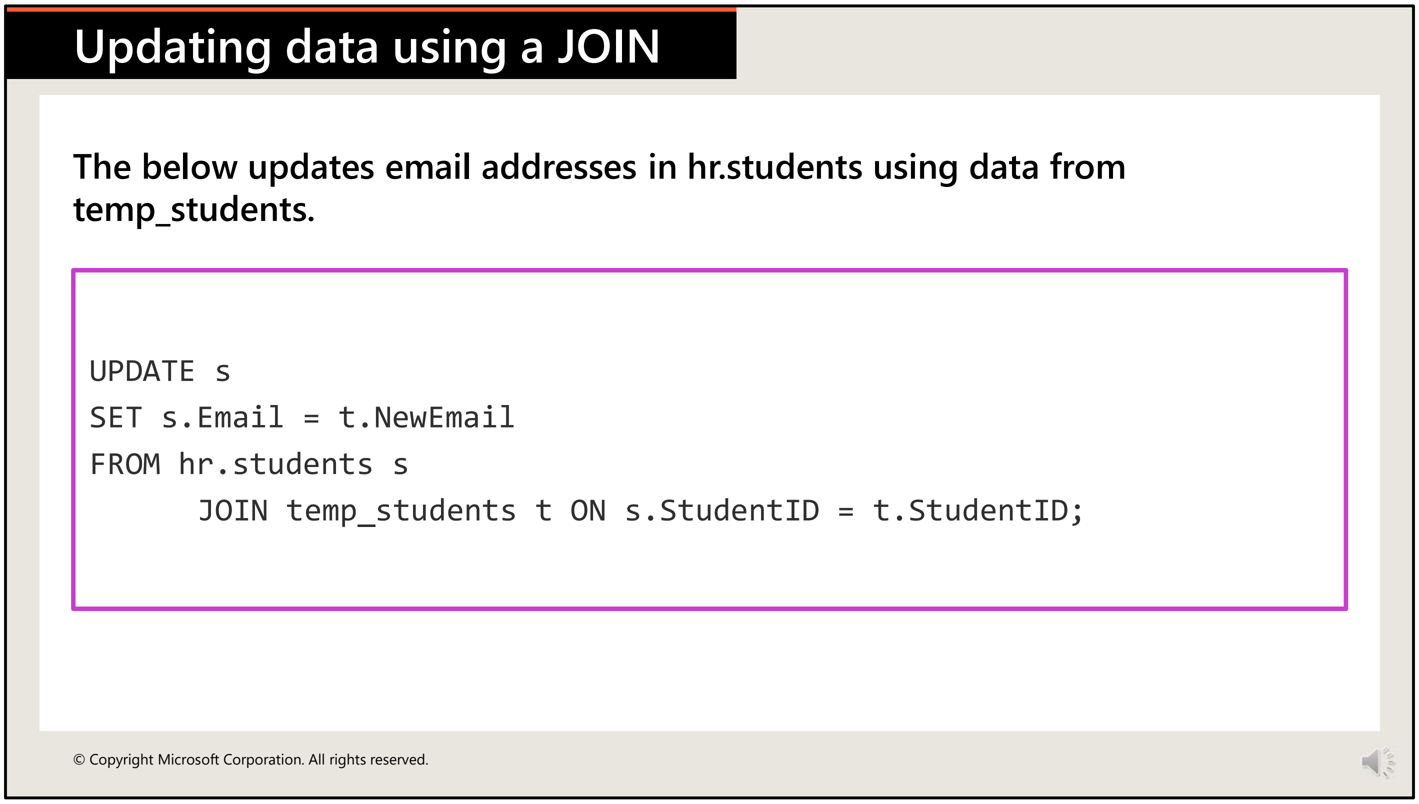
Updating data in a table

Updates all rows in a table or view

- Set can be filtered with a WHERE clause
- Set can be defined with a FROM clause

Only columns specified in the SET clause are modified

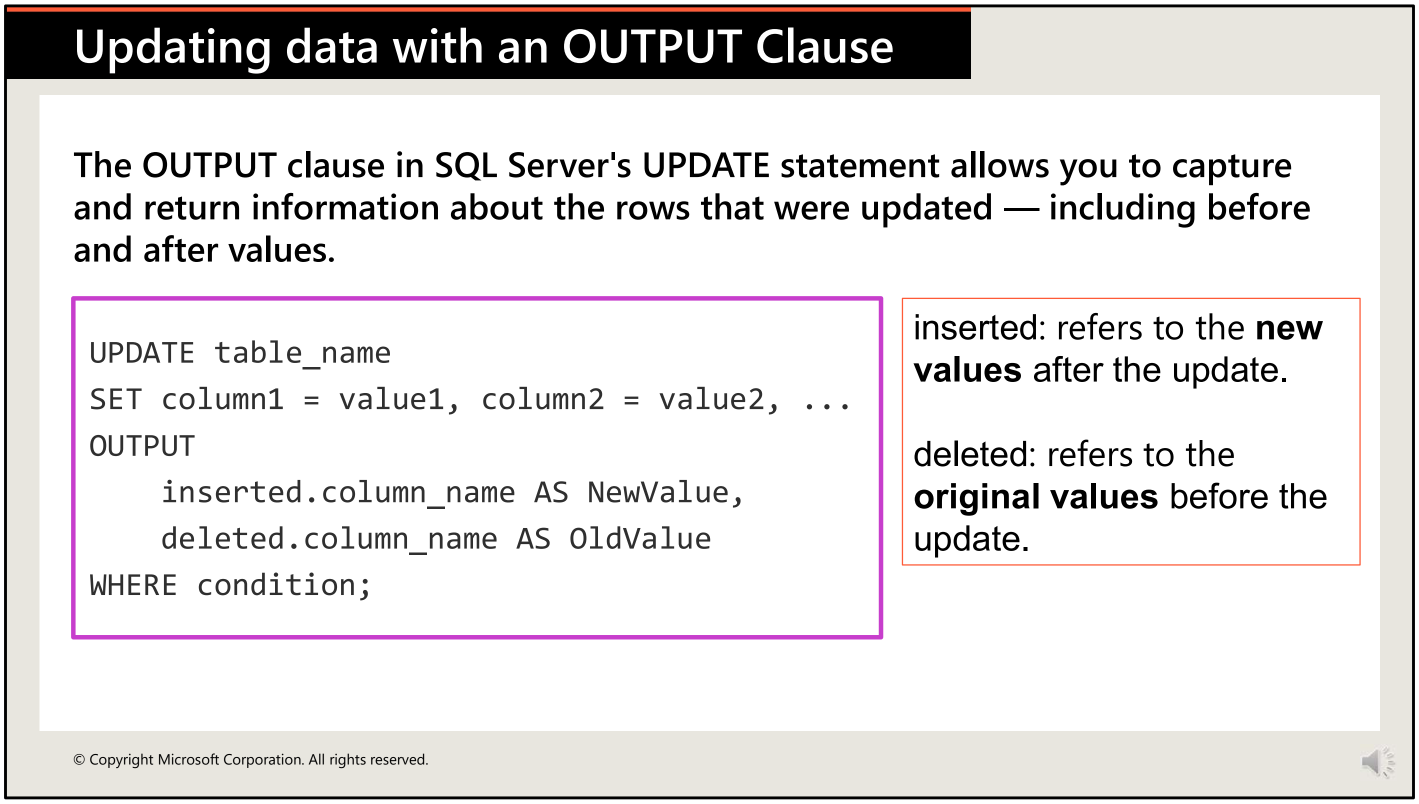
```
UPDATE Sales.Promotion
SET Notes = '25% off socks'
WHERE PromotionID = 2;
```



Updating data using a JOIN

The below updates email addresses in hr.students using data from temp_students.

```
UPDATE s
SET s.Email = t.NewEmail
FROM hr.students s
      JOIN temp_students t ON s.StudentID = t.StudentID;
```



Updating data with an OUTPUT Clause

The OUTPUT clause in SQL Server's UPDATE statement allows you to capture and return information about the rows that were updated — including before and after values.

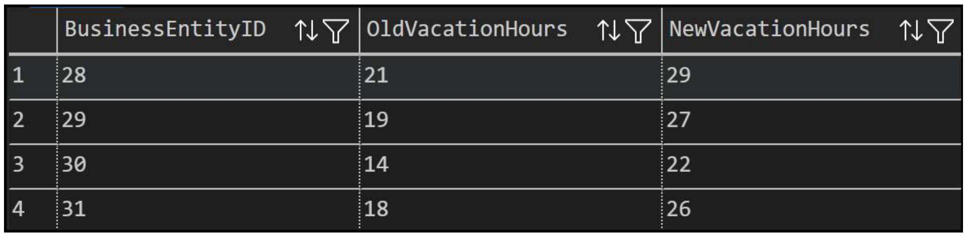
```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
OUTPUT
    inserted.column_name AS NewValue,
    deleted.column_name AS OldValue
WHERE condition;
```

inserted: refers to the **new values** after the update.

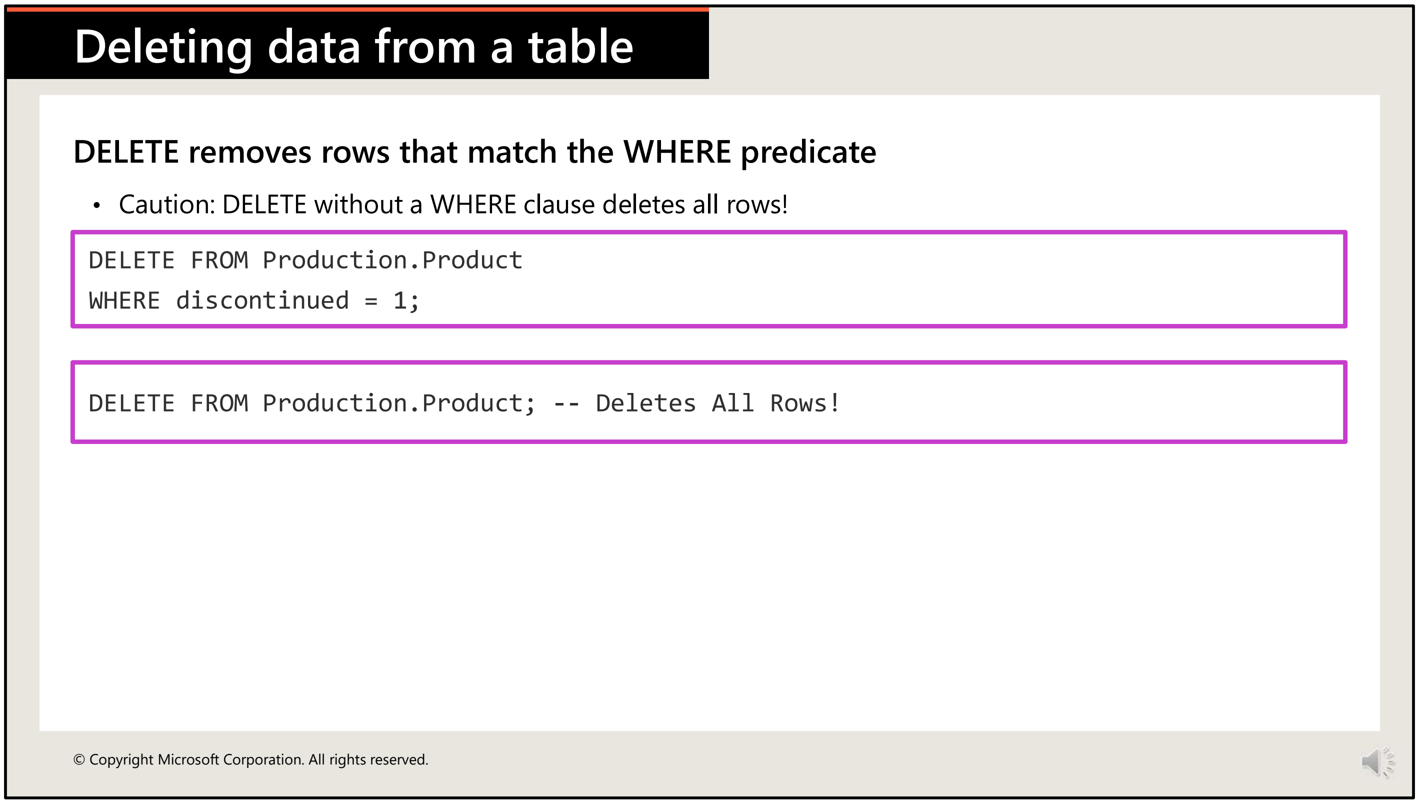
deleted: refers to the **original values** before the update.

Updating data with an OUTPUT Clause

```
UPDATE HumanResources.Employee
SET VacationHours = VacationHours + 8
OUTPUT
    inserted.BusinessEntityID,
    deleted.VacationHours AS OldVacationHours,
    inserted.VacationHours AS NewVacationHours
WHERE JobTitle = 'Production Technician - WC60';
```



	BusinessEntityID	OldVacationHours	NewVacationHours
1	28	21	29
2	29	19	27
3	30	14	22
4	31	18	26



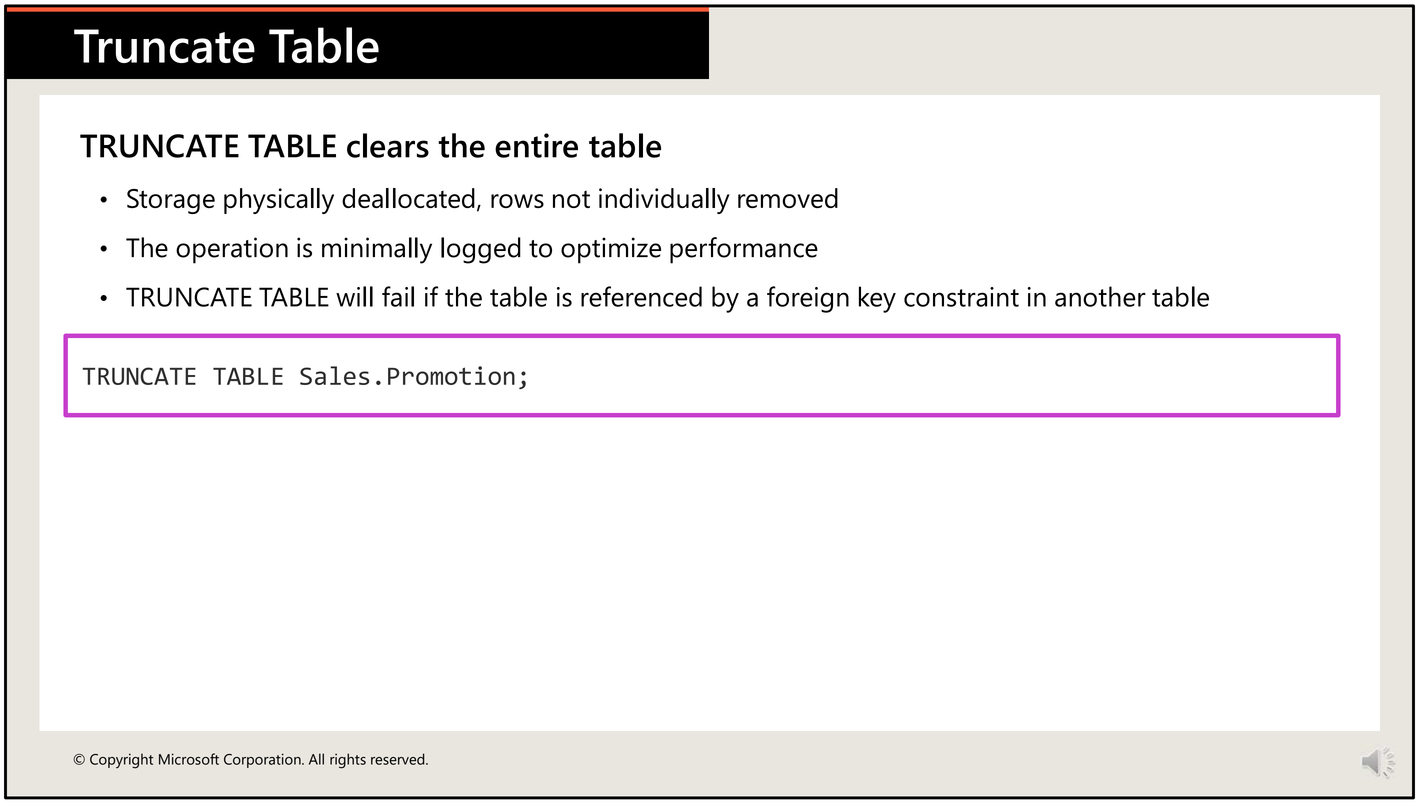
Deleting data from a table

DELETE removes rows that match the WHERE predicate

- Caution: DELETE without a WHERE clause deletes all rows!

```
DELETE FROM Production.Product
WHERE discontinued = 1;
```

```
DELETE FROM Production.Product; -- Deletes All Rows!
```

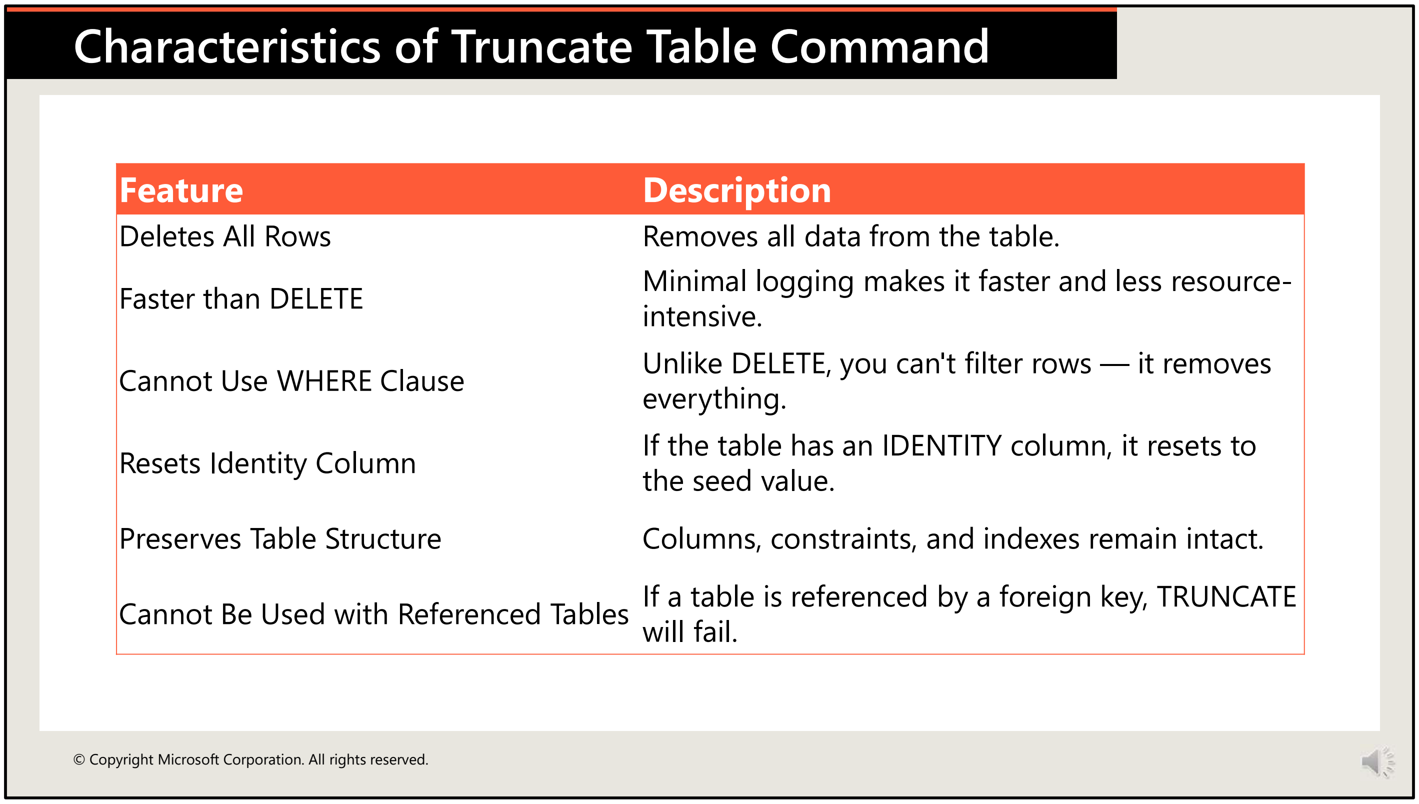


Truncate Table

TRUNCATE TABLE clears the entire table

- Storage physically deallocated, rows not individually removed
- The operation is minimally logged to optimize performance
- TRUNCATE TABLE will fail if the table is referenced by a foreign key constraint in another table

```
TRUNCATE TABLE Sales.Promotion;
```



Characteristics of Truncate Table Command

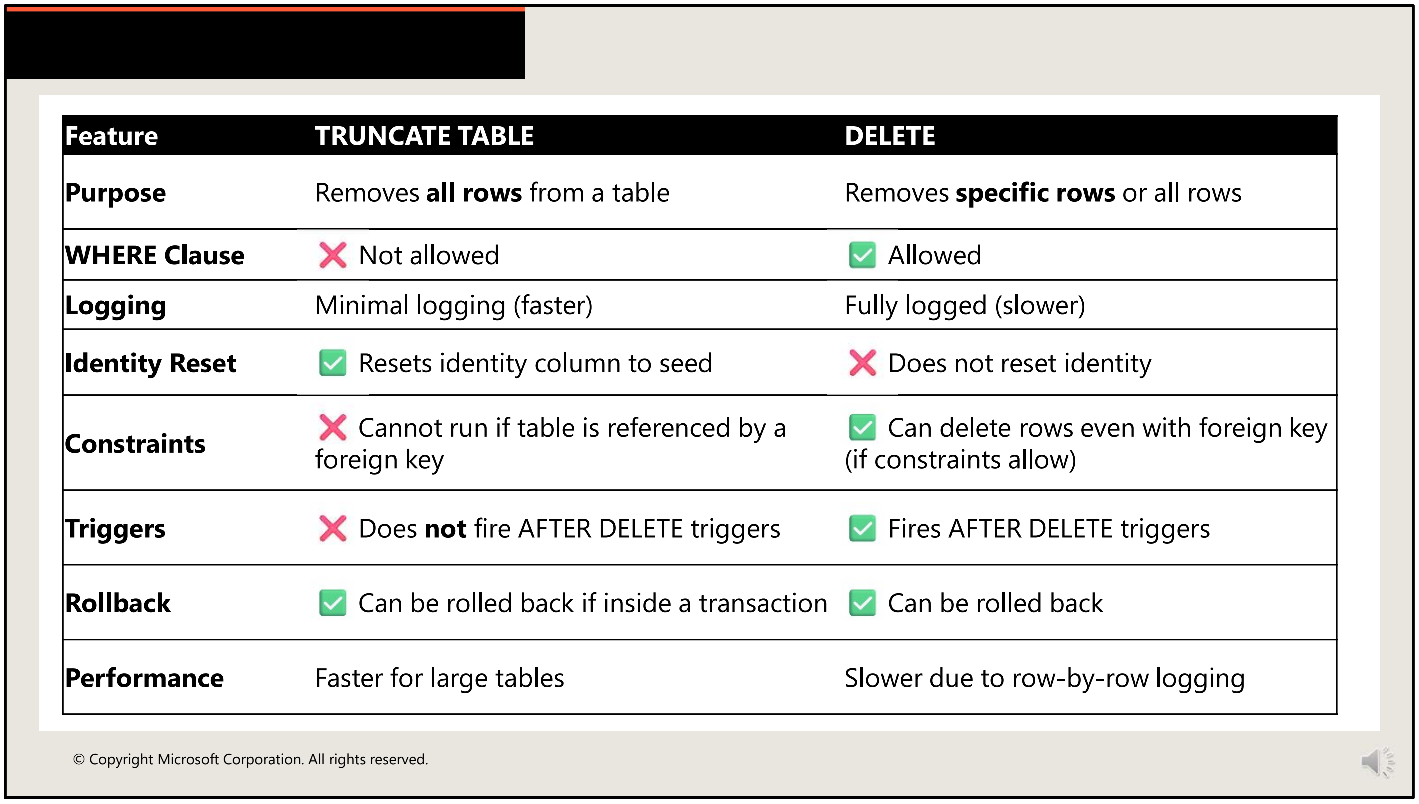
Feature	Description
Deletes All Rows	Removes all data from the table.
Faster than DELETE	Minimal logging makes it faster and less resource-intensive.
Cannot Use WHERE Clause	Unlike DELETE, you can't filter rows — it removes everything.
Resets Identity Column	If the table has an IDENTITY column, it resets to the seed value.
Preserves Table Structure	Columns, constraints, and indexes remain intact.
Cannot Be Used with Referenced Tables	If a table is referenced by a foreign key, TRUNCATE will fail.

Removes all rows from a table, without logging the individual row deletions. TRUNCATE TABLE is similar to the DELETE statement with no WHERE clause; however, TRUNCATE TABLE is faster and uses fewer system and transaction log resources.

Compared to the DELETE statement, TRUNCATE TABLE has the following advantages:

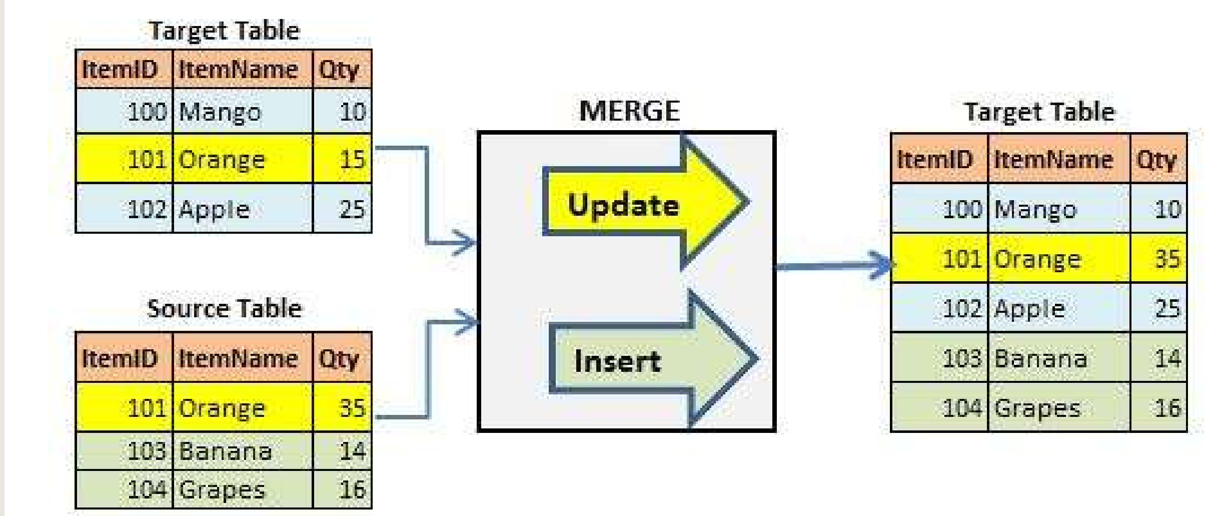
- Less transaction log space is used: The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row. TRUNCATE TABLE removes the data by deallocating the data pages used to store the table and index data and records only the page deallocations in the transaction log.

- TRUNCATE TABLE removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on, remain. To remove the table definition in addition to its data, use the DROP TABLE statement.



Feature	TRUNCATE TABLE	DELETE
Purpose	Removes **all rows** from a table	Removes **specific rows** or all rows
WHERE Clause	❌ Not allowed	✅ Allowed
Logging	Minimal logging (faster)	Fully logged (slower)
Identity Reset	✅ Resets identity column to seed	❌ Does not reset identity
Constraints	❌ Cannot run if table is referenced by a foreign key	✅ Can delete rows even with foreign key (if constraints allow)
Triggers	❌ Does **not** fire AFTER DELETE triggers	✅ Fires AFTER DELETE triggers
Rollback	✅ Can be rolled back if inside a transaction	✅ Can be rolled back
Performance	Faster for large tables	Slower due to row-by-row logging

Merging data in a table



Target Table

ItemID	ItemName	Qty
100	Mango	10
101	Orange	15
102	Apple	25

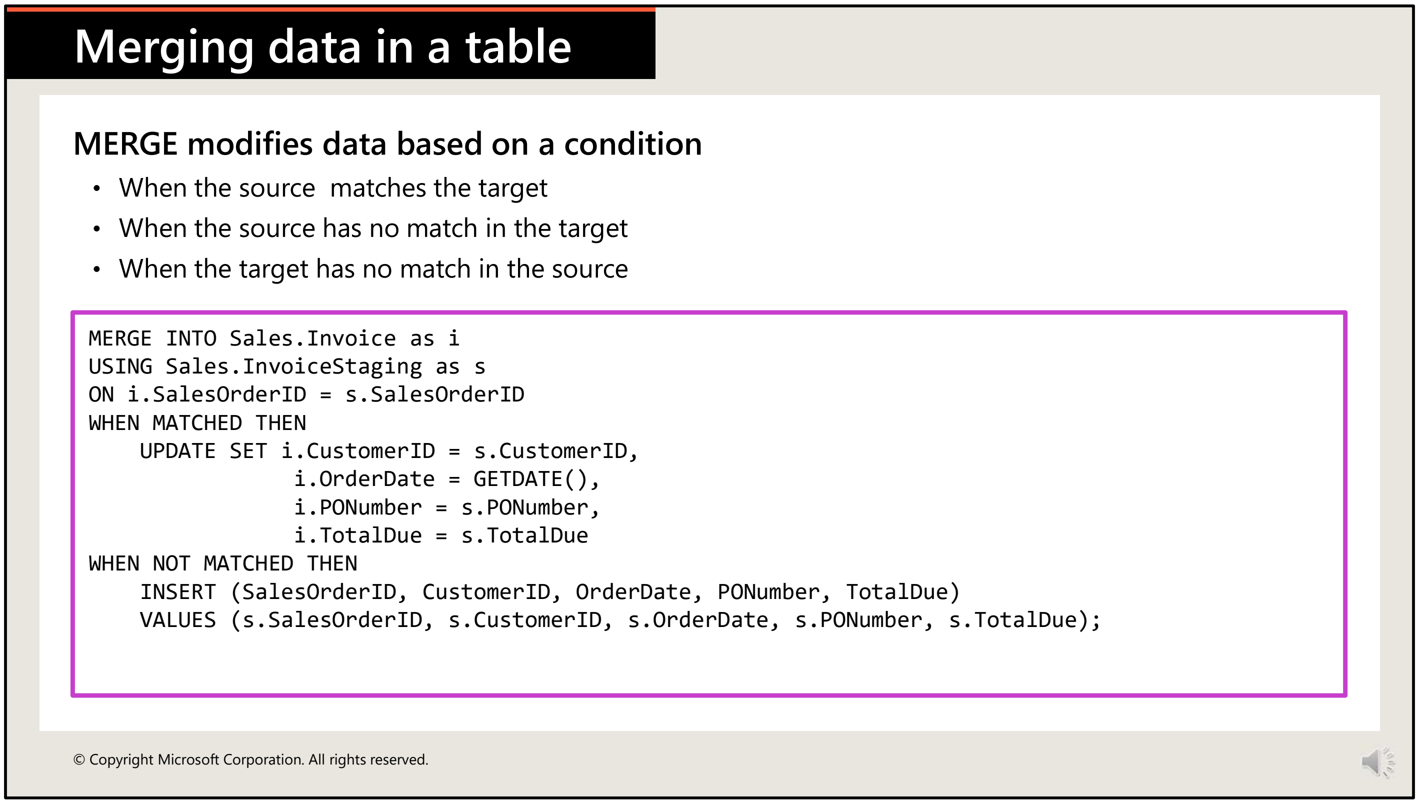
Source Table

ItemID	ItemName	Qty
101	Orange	35
103	Banana	14
104	Grapes	16

Target Table

ItemID	ItemName	Qty
100	Mango	10
101	Orange	35
102	Apple	25
103	Banana	14
104	Grapes	16

Ref: https://www.sql-datatools.com/2015/10/merge-statement-in-sql-server.html



Merging data in a table

MERGE modifies data based on a condition

- When the source matches the target
- When the source has no match in the target
- When the target has no match in the source

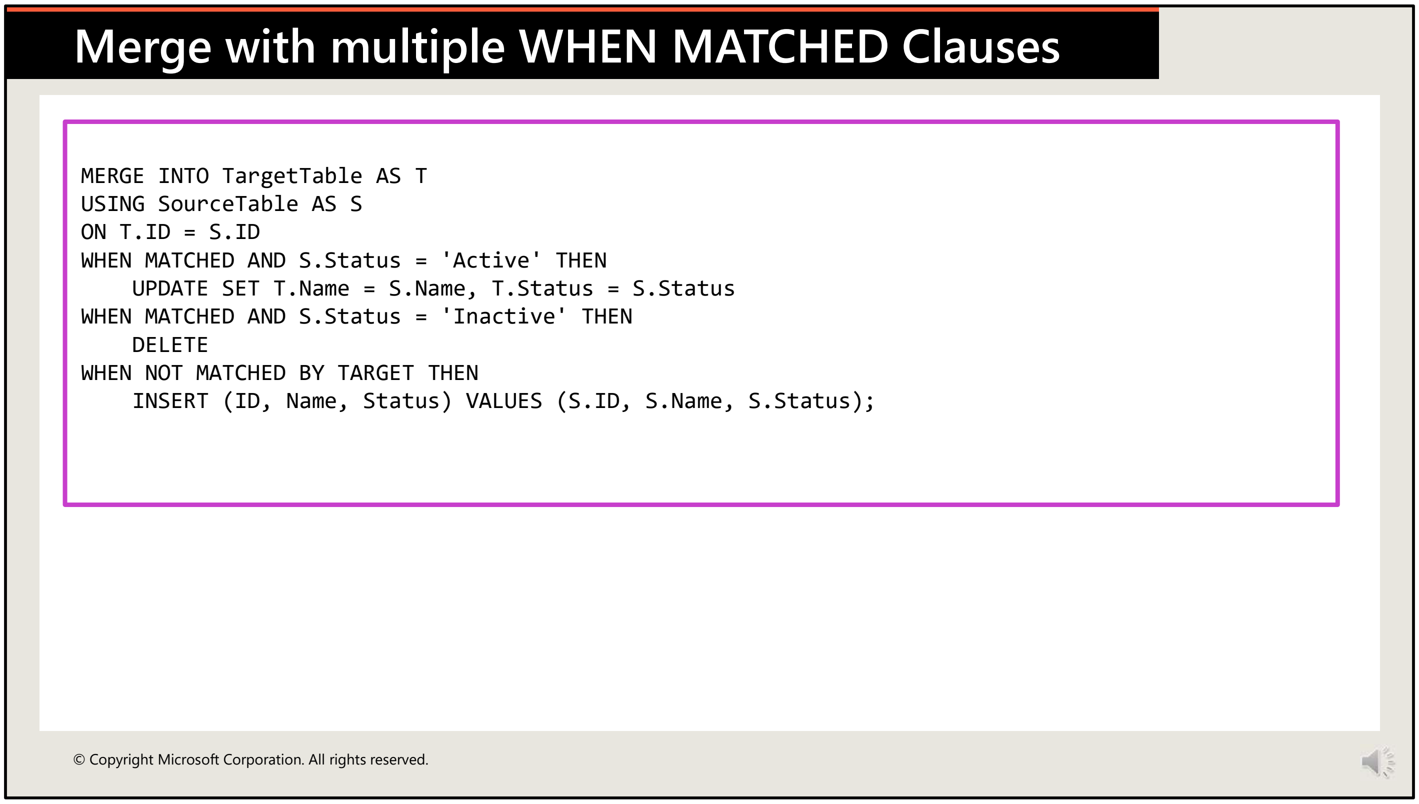
```
MERGE INTO Sales.Invoice as i
USING Sales.InvoiceStaging as s
ON i.SalesOrderID = s.SalesOrderID
WHEN MATCHED THEN
    UPDATE SET i.CustomerID = s.CustomerID,
                i.OrderDate = GETDATE(),
                i.PONumber = s.PONumber,
                i.TotalDue = s.TotalDue
WHEN NOT MATCHED THEN
    INSERT (SalesOrderID, CustomerID, OrderDate, PONumber, TotalDue)
    VALUES (s.SalesOrderID, s.CustomerID, s.OrderDate, s.PONumber, s.TotalDue);
```

The MERGE statement runs insert, update, or delete operations on a target table from the results of a join with a source table. For example, synchronize two tables by inserting, updating, or deleting rows in one table based on differences found in the other table.

The MERGE statement lets you insert, update, or delete data in a target table based on matching records in a source table—all in one command. It's often called an "upsert" (update or insert).

Motivation:

- Efficiency: Handle multiple scenarios in a single statement.
- Clarity: Logic is all in one place—easier to maintain.
- Atomicity: All changes happen together, reducing risk of inconsistent data.



Merge with multiple WHEN MATCHED Clauses

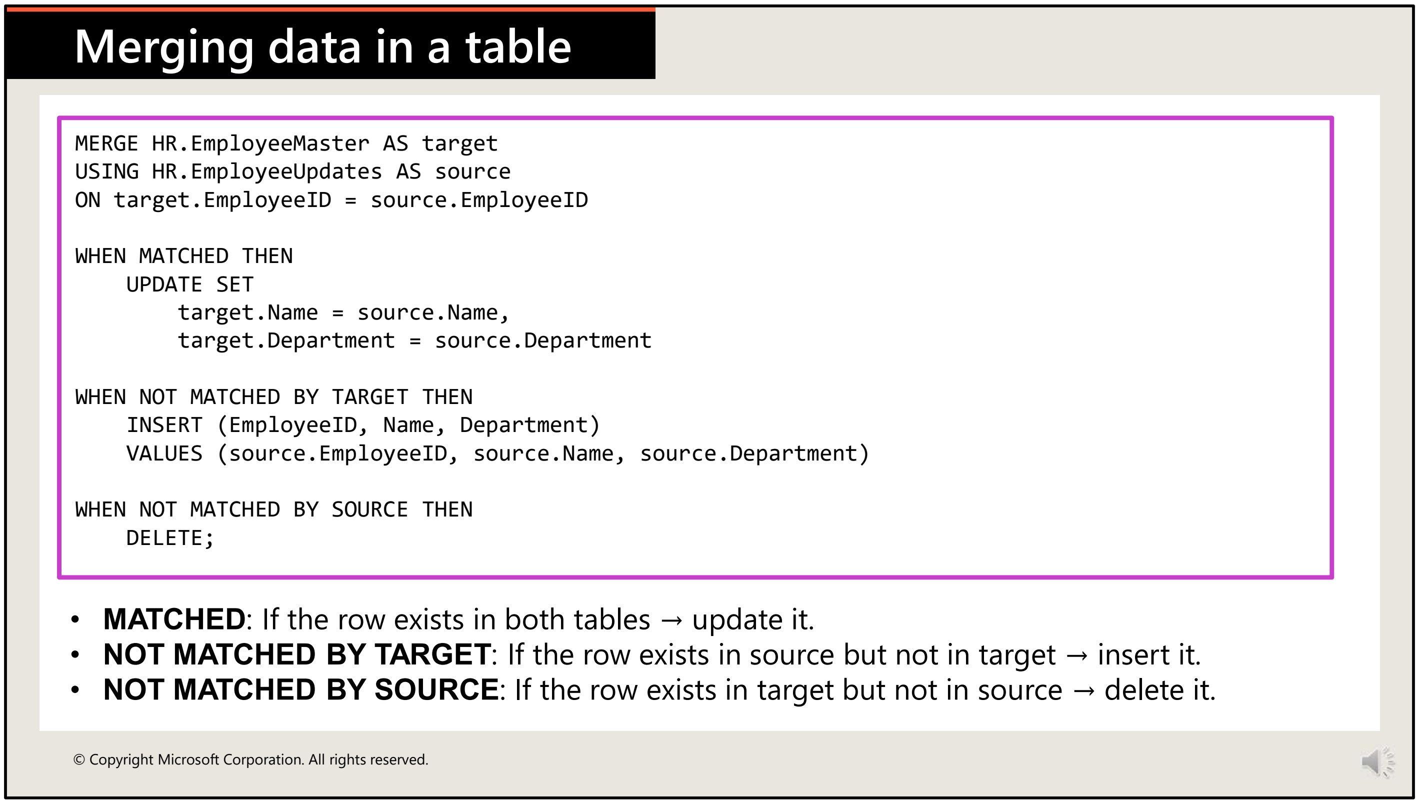
```
MERGE INTO TargetTable AS T
USING SourceTable AS S
ON T.ID = S.ID
WHEN MATCHED AND S.Status = 'Active' THEN
    UPDATE SET T.Name = S.Name, T.Status = S.Status
WHEN MATCHED AND S.Status = 'Inactive' THEN
    DELETE
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, Name, Status) VALUES (S.ID, S.Name, S.Status);
```

It can have up to two WHEN MATCHED clauses. Sometimes, you want to perform different actions depending on the details of the match. For example, you might want to update one set of rows one way, and other set of rows a different way, based on a condition.

First WHEN MATCHED: If a row matches and Status is 'Active', update the target.
Second WHEN MATCHED: If a row matches and Status is 'Inactive', delete the row from the target.
WHEN NOT MATCHED BY TARGET: If a row exists in the source but not in the target, insert it.

If two clauses are specified, the first clause must be accompanied by an AND <search_condition> clause. For any given row, the second WHEN MATCHED clause is only applied if the first isn't.

If there are two WHEN MATCHED clauses, one must specify an UPDATE action and one must specify a DELETE action.

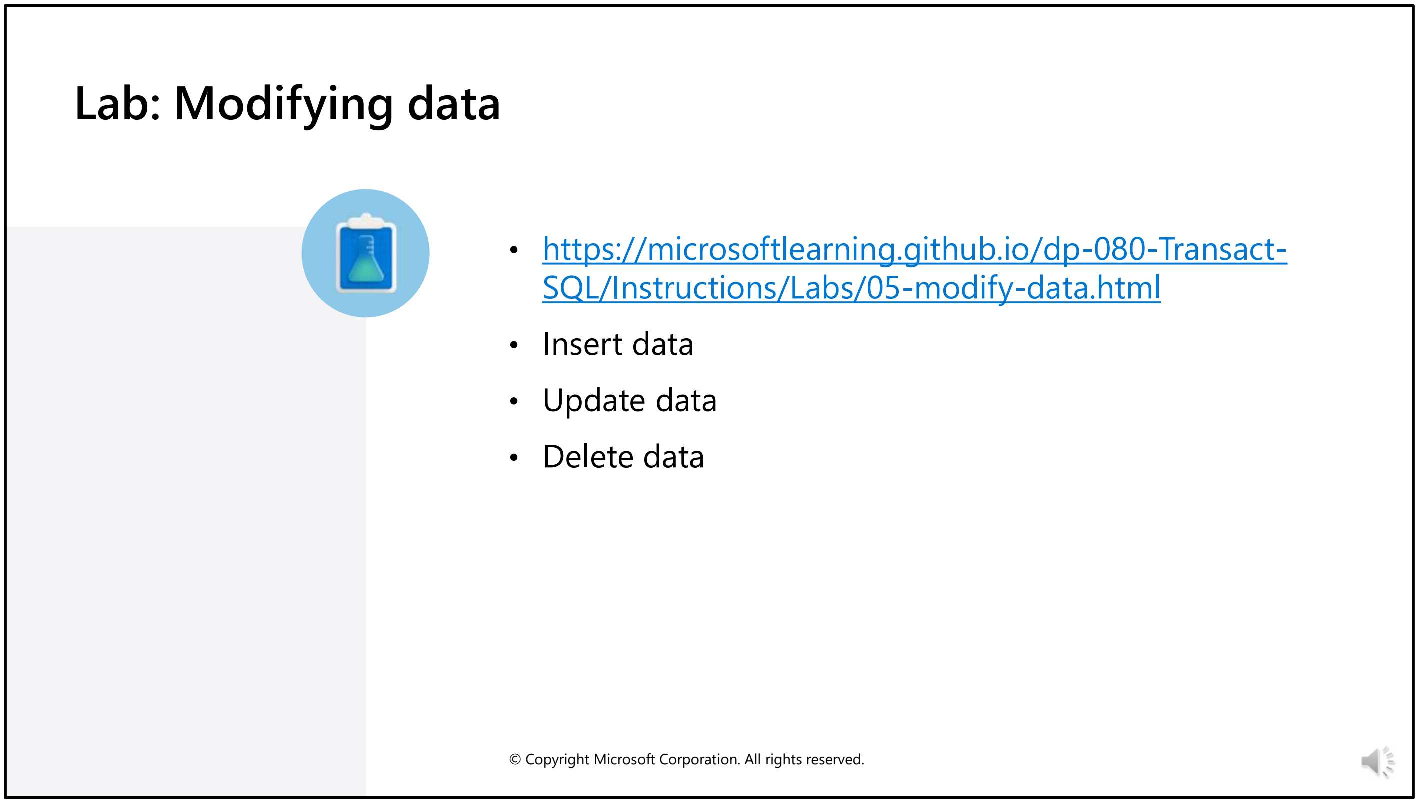


Merging data in a table

```
MERGE HR.EmployeeMaster AS target
USING HR.EmployeeUpdates AS source
ON target.EmployeeID = source.EmployeeID

WHEN MATCHED THEN
    UPDATE SET
        target.Name = source.Name,
        target.Department = source.Department

WHEN NOT MATCHED BY TARGET THEN
    INSERT (EmployeeID, Name, Department)
    VALUES (source.EmployeeID, source.Name, source.Department)

WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

- **MATCHED**: If the row exists in both tables → update it.
- **NOT MATCHED BY TARGET**: If the row exists in source but not in target → insert it.
- **NOT MATCHED BY SOURCE**: If the row exists in target but not in source → delete it.

When UPDATE is specified in the <merge_matched> clause, and more than one row of <table_source> matches a row in *target_table* based on <merge_search_condition>, SQL Server returns an error. The MERGE statement can't update the same row more than once, or update and delete the same row.



Lab: Modifying data

- https://microsoftlearning.github.io/dp-080-Transact-SQL/Instructions/Labs/05-modify-data.html
- Insert data
- Update data
- Delete data

https://microsoftlearning.github.io/dp-080-Transact-SQL/Instructions/Labs/05-modify-data.html

Microsoft