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

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 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:

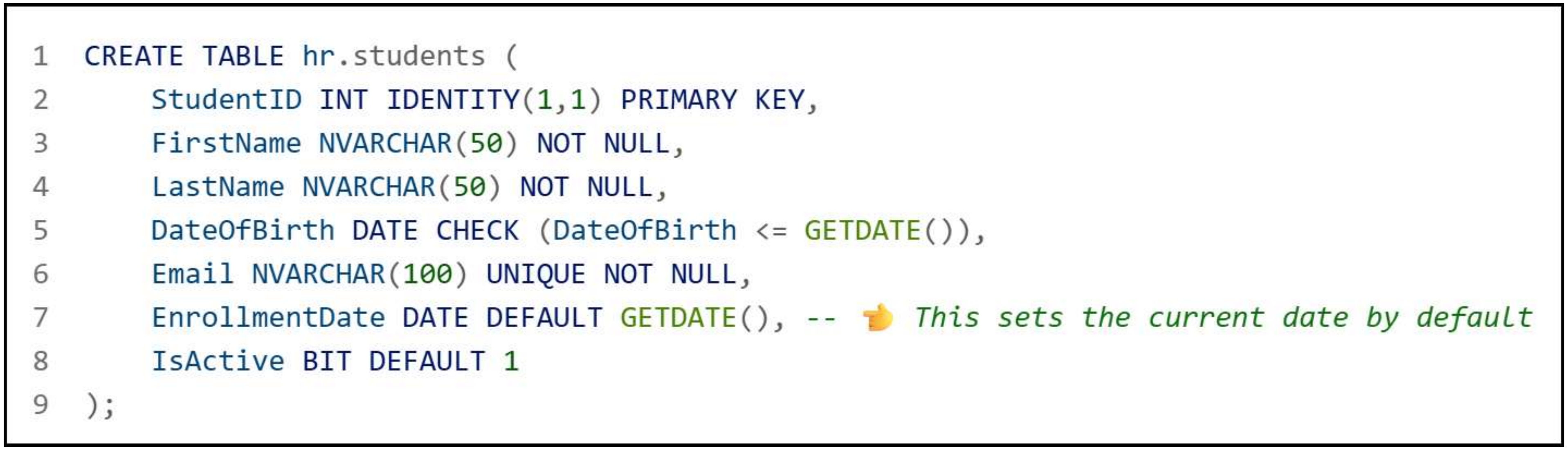
  @@IDENTITY: The last identity generated in the session

  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
);
```

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

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)
```

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’);
```

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);
```

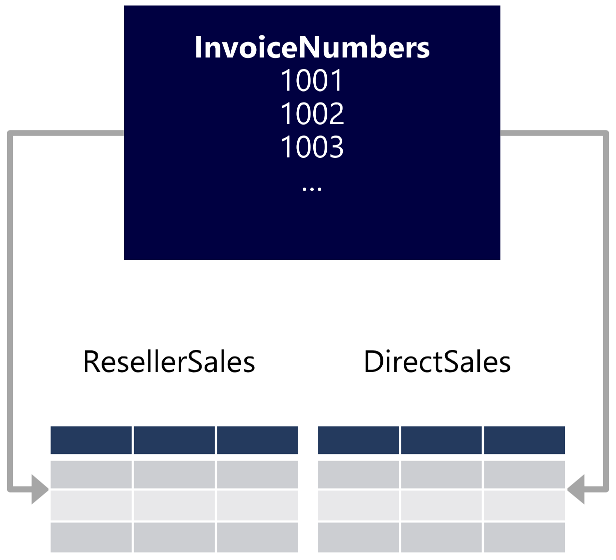
```
ALTER TABLE hr.students_backup
ADD CONSTRAINT UQ_students_backup_Email UNIQUE (Email);
```

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;
```

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.

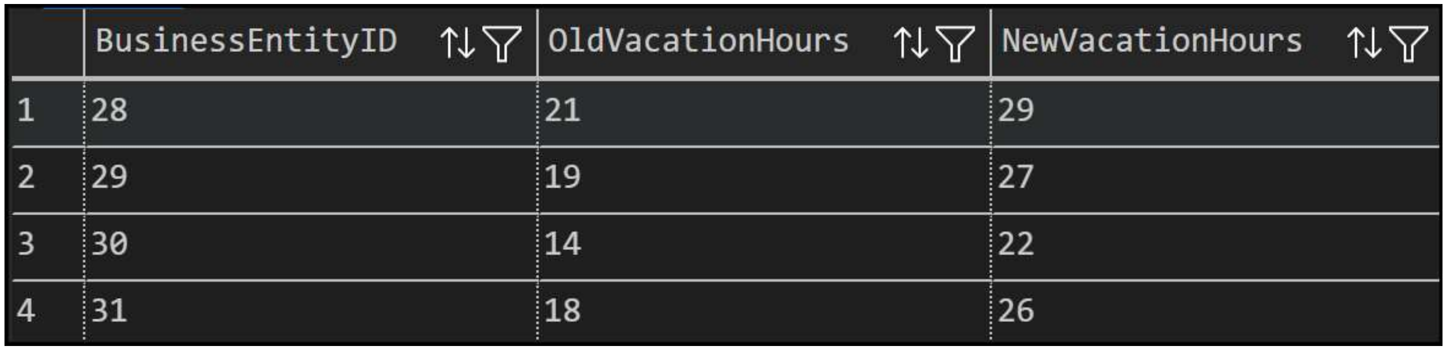
```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
OUTPUT
    inserted.column_name AS NewValue,
    deleted.column_name AS OldValue
WHERE condition;
```

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

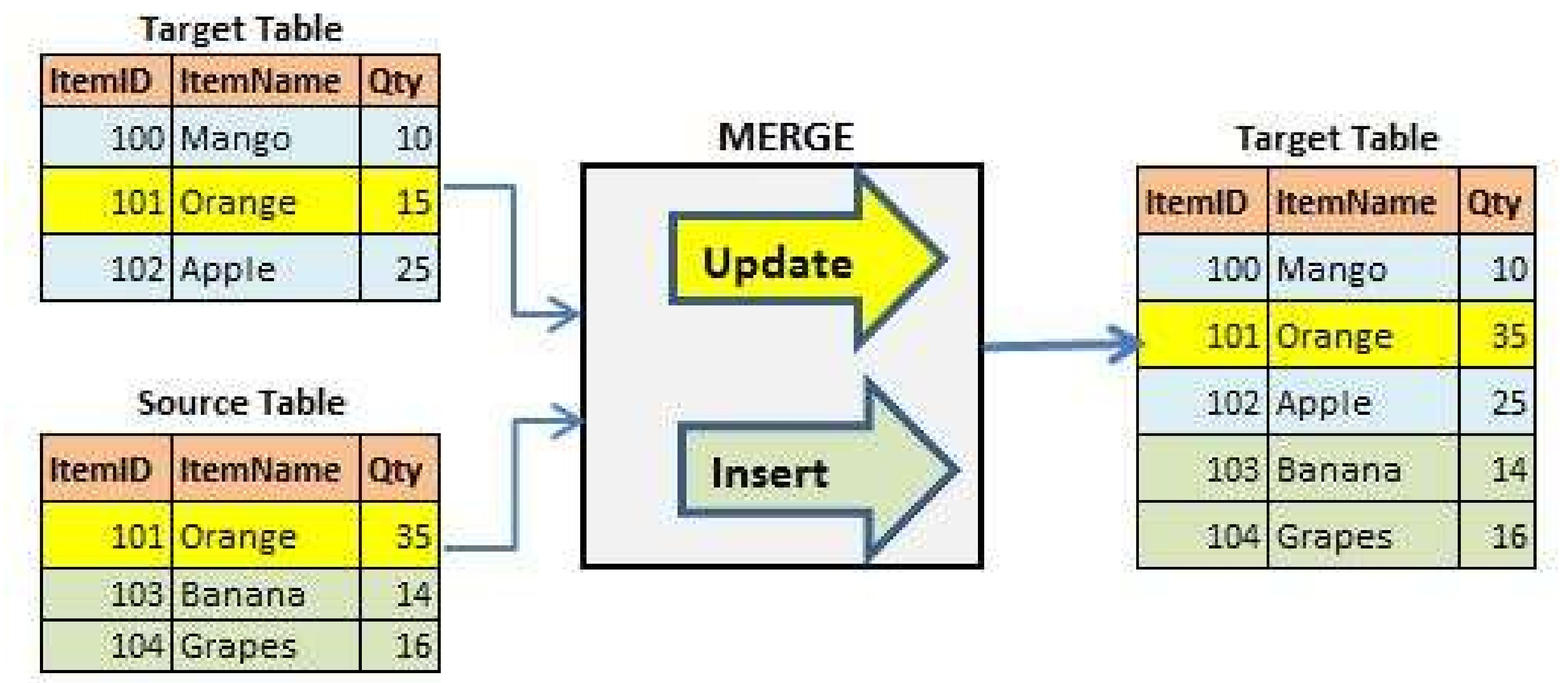
```
TRUNCATE TABLE Sales.Promotion;
```

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.

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

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);
```

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);
```

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.

Lab: Modifying data

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

Microsoft