Using INSERT to Add Data

- The INSERT…VALUES statement inserts a new row

```
INSERT INTO Sales.OrderDetails
       (orderid, productid, unitprice, qty, discount)
VALUES (10255,39,18,2,0.05);
```

- Table and row constructors add multi-row capability to INSERT…VALUES

```
INSERT INTO Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)

VALUES
(10256,39,18,2,0.05),
(10258,39,18,5,0.10);
```

Using INSERT with Data Providers

- INSERT … SELECT to insert rows from another table:

```
INSERT Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)

        SELECT * FROM NewOrderDetails
```

```
INSERT [INTO] <table or view> [(column_list)]

        SELECT <column_list> FROM <table_list> …;
```

Using SELECT INTO

SELECT -> INTO is similar to INSERT <- SELECT

- It also creates a table for the output, fashioned on the output itself

- The new table is based on query column structure
  - Uses column names, data types, and null settings
  - Does not copy constraints or indexes

```
SELECT *
       INTO NewProducts
FROM PRODUCTION.PRODUCTS
WHERE ProductID >= 70
```

INSERT (overview)

```
INSERT INTO Sales.OrderDetails
       (orderid, productid, unitprice, qty, discount)
VALUES (10255,39,18,2,0.05);
```

```
INSERT INTO Sales.OrderDetails
       (orderid, productid, unitprice, qty, discount)
VALUES
       (10256,39,18,2,0.05),
       (10258,39,18,5,0.10);
```

```
INSERT INTO Sales.OrderDetails
       (orderid, productid, unitprice, qty, discount)
    SELECT * FROM NewOrderDetails
```

```
SELECT *
       INTO NewProducts
FROM PRODUCTION.PRODUCTS
WHERE ProductID >= 70
```

Using UPDATE to Modify Data

- UPDATE changes all rows in a table or view
  - Unless rows are filtered with a WHERE clause or constrained with a JOIN clause
- Column values are changed with the SET clause

```
UPDATE Production.Products
   SET   unitprice = (unitprice * 1.04)
WHERE    categoryid =  1 AND discontinued = 0
;
```

```
UPDATE Production.Products
   SET     unitprice *= 1.04
              -- Using compound assignment operators
WHERE    categoryid =  1 AND discontinued = 0;
```

Using DELETE to Remove Data

- DELETE:
  removes all rows from the target table that meet the condition defined in a WHERE clause.

```
DELETE FROM  Sales.OrderDetails
WHERE   orderid =  10248;
```

Using TRUNCATE TABLE to Remove Data

```
TRUNCATE TABLE Sales.OrderDetails;
```

- TRUNCATE TABLE:

  removes all rows from the target table

  - does not support a WHERE clause to restrict which rows are deleted
  - fast: uses less space in the transaction log than DELETE, since DELETE logs individual row deletions, while TRUNCATE TABLE only logs the deallocation of storage space
  - cannot be used on a table with a foreign key reference to another
  - TRUNCATE TABLE operation can be rolled back and all rows restored if TRUNCATE is issued within a user-defined transaction

Using OUTPUT Clause

- returns information from each row affected by an INSERT, UPDATE, DELETE, or MERGE statement.

```
INSERT INTO HR.Employees
(
Title, titleofcourtesy,
FirstName, Lastname, hiredate, birthdate,
address, city, country, phone
)

OUTPUT INSERTED.*

VALUES
(
'Sales Representative', 'Mr',
'Laurence', 'Grider', '04/04/2016', '10/25/1975',
'1234 1st Ave. S.E.', 'Seattle', 'USA', '(206)555-0105'
);
```

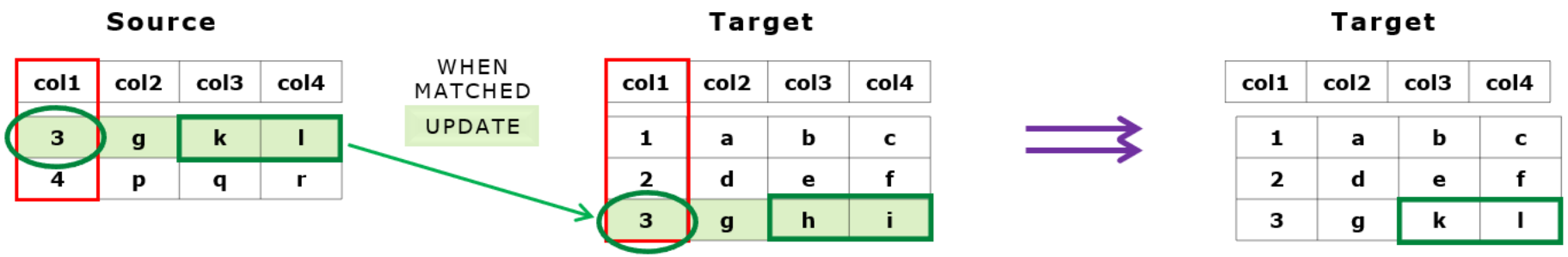
Results | Messages

	empid	lastname	firstname	title	titleofcourtesy	birthdate	hiredate	address	city
1	11	Grider	Laurence	Sales Representative	Mr	1975-10-25 00:00:00.000	2016-04-04 00:00:00.000	1234 1st Ave. S.E.	Seattle

Using MERGE to Modify Data

MERGE modifies data based on a condition

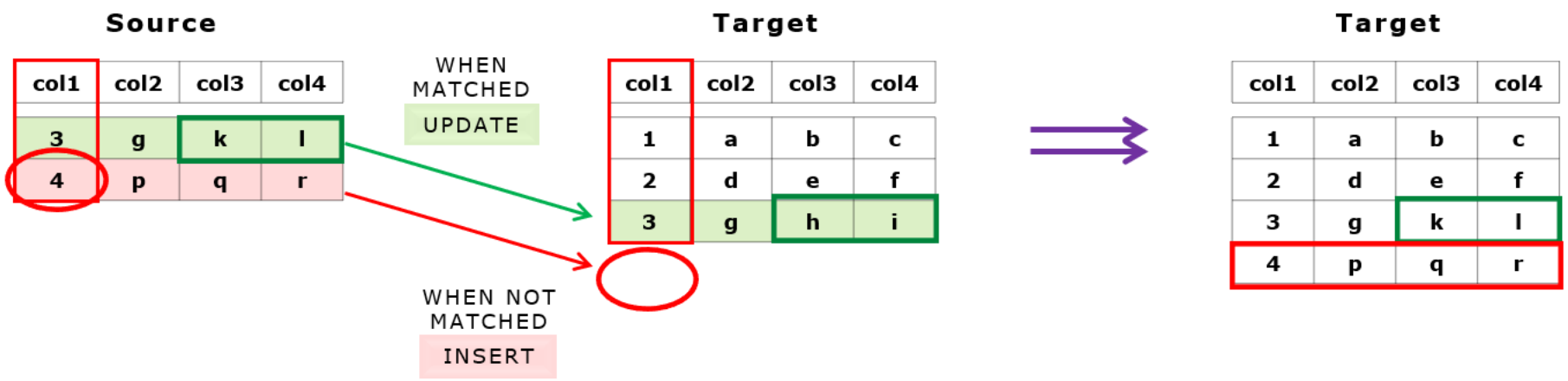
- When the source matches the target



Using MERGE to Modify Data

MERGE modifies data based on a condition

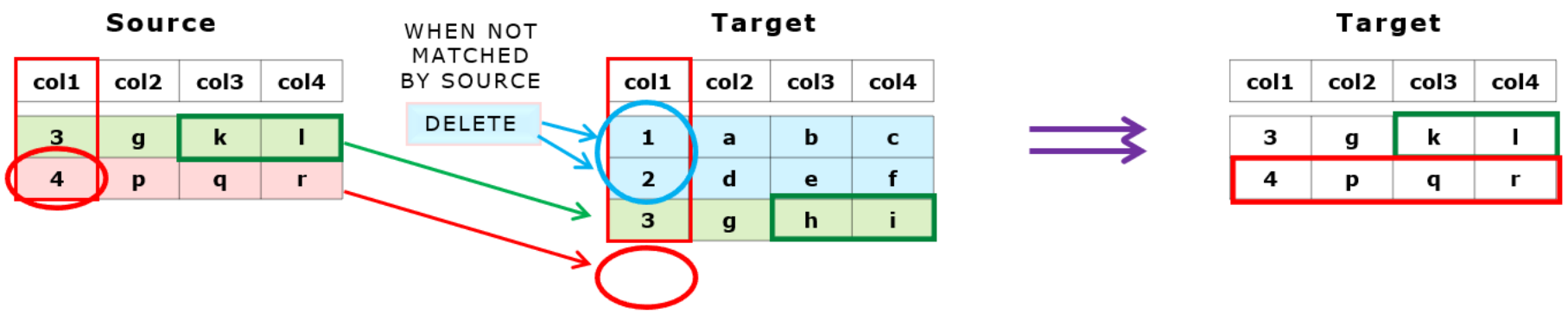
- When the source matches the target
- When the source has no match in the target



Using MERGE to Modify Data

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 Statement

Common Table Expression

- Modifies data in a target table (or updatable view or CTE) based on the results of a join with a source table
- Commonly used to populate data warehouses
  - INSERT data if not already present
  - UPDATE data if already present
- Target table plus a source rowset
- Must specify how the source and target are joined

Target Table: is being modified

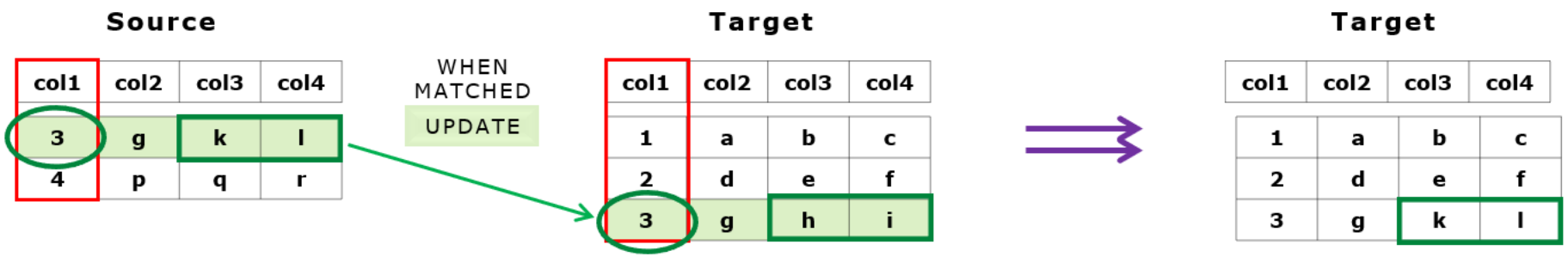
SourceTable: incoming data

```
MERGE INTO dbo.Employee AS e
USING dbo.EmployeeUpdate AS eu
ON e.EmployeeID = eu.EmployeeID
...
```

Using MERGE to Modify Data

MERGE modifies data based on a condition

- When the source matches the target



WHEN MATCHED

- Clause that defines the action to be taken when the row in the source is found in the target
- Specifies the data modifications to take place – can be INSERT, UPDATE or DELETE
- Two WHEN MATCHED clauses can be included – needs an extra predicate on the first
  - WHEN MATCHED AND s.Quantity > 0
  - One must be an UPDATE, the other a DELETE

```
IF matched1 THEN ..........
  ELSE matched2: ..........
```

```
MERGE INTO dbo.Employee AS e
USING dbo.EmployeeUpdate AS eu
ON e.EmployeeID = eu.EmployeeID
WHEN MATCHED THEN
  UPDATE SET e.FullName = eu.FullName,
             e.EmploymentStatus = eu.EmploymentStatus
```

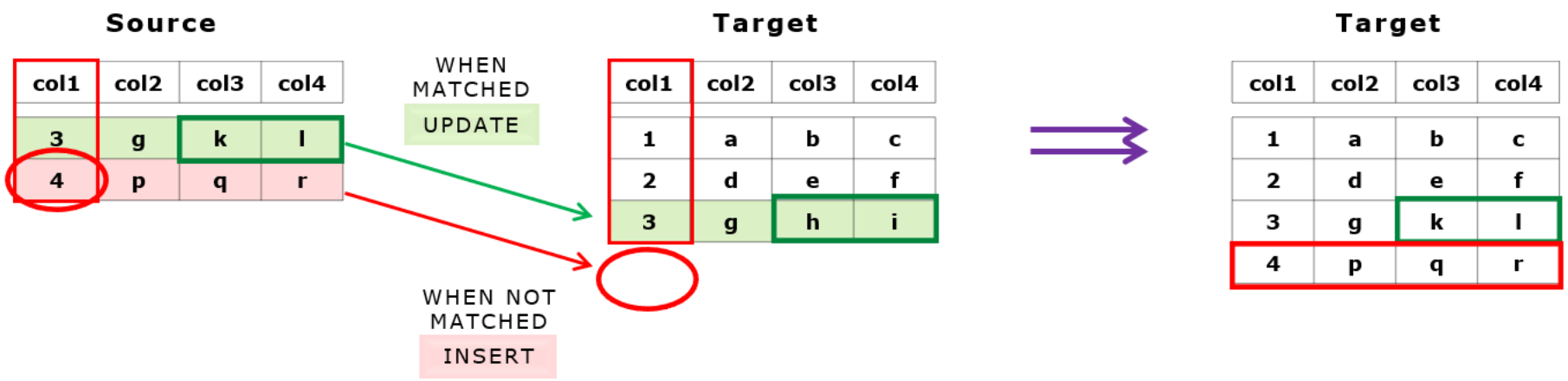
Target Table: is being modified

SourceTable: incoming data

Using MERGE to Modify Data

MERGE modifies data based on a condition

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



WHEN NOT MATCHED BY TARGET

- Clause that defines the action to be taken when the row in the source cannot be found in the target
- The words BY TARGET are optional and often omitted

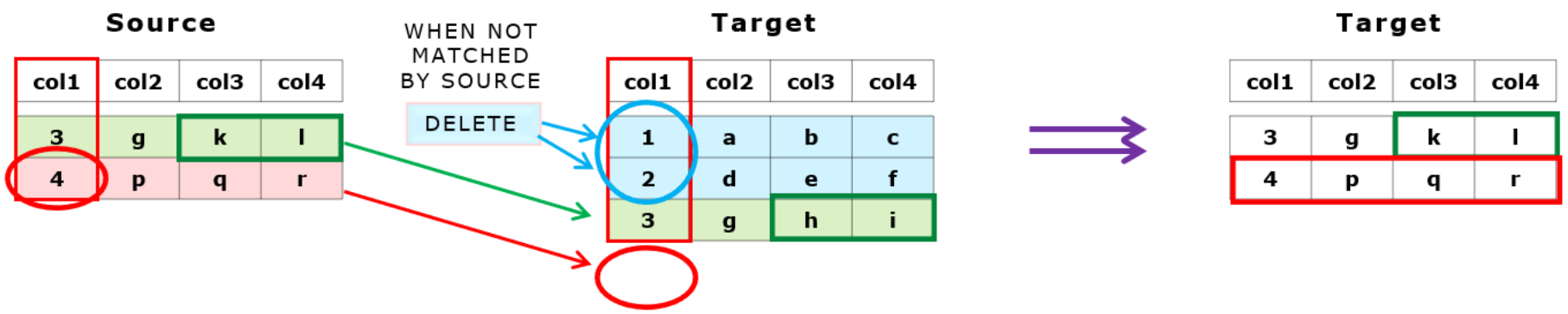
```
MERGE INTO dbo.Employee AS e
USING dbo.EmployeeUpdate AS eu
ON e.EmployeeID = eu.EmployeeID
WHEN MATCHED THEN
 UPDATE SET e.FullName = eu.FullName,
            e.EmploymentStatus = eu.EmploymentStatus
WHEN NOT MATCHED THEN
 INSERT (EmployeeID,FullName,EmploymentStatus)
 VALUES
    (eu.EmployeeID,eu.FullName,eu.EmploymentStatus);
```

INSERT [column list], n.b.: column list is optional

Using MERGE to Modify Data

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



WHEN NOT MATCHED BY SOURCE

- Clause that defines the action to be taken for rows in the target that were not supplied in the source
  - Not commonly used but typically involve a DELETE

```
MERGE INTO dbo.Employee AS e
USING dbo.EmployeeUpdate AS eu
ON e.EmployeeID = eu.EmployeeID
WHEN MATCHED THEN
 UPDATE SET e.FullName = eu.FullName,
            e.EmploymentStatus = eu.EmploymentStatus
WHEN NOT MATCHED THEN
 INSERT (EmployeeID,FullName,EmploymentStatus)
 VALUES
     (eu.EmployeeID,eu.FullName,eu.EmploymentStatus)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

N.B.: the DELETE statement has no table or predicate specified

Using IDENTITY

The IDENTITY property generates column values automatically

- Optional seed and increment values can be provided

```
CREATE TABLE Production.Products
(PID int IDENTITY(1,1) NOT NULL, Name VARCHAR(15),…)
```

- Only one column in a table may have IDENTITY defined
- IDENTITY column must be omitted in a normal INSERT statement

```
INSERT INTO Production.Products (Name,…)
VALUES ('MOC 2072 Manual',…)
```

- There is a setting to allow identity columns to be changed manually ON or automatic OFF
  - SET IDENTITY_INSERT <Tablename> [ON|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;
```

SCOPE_IDENTITY function to retrieve the most recent *identity* value that has been assigned in the database (to any table),

the **IDENT_CURRENT** function, which retrieves the latest *identity* value in the specified table.

Using Sequences

Sequence objects were first added in SQL Server 2012

- Independent objects in database
  - More flexible than the IDENTITY property
  - Can be used as default value for a column

- Manage with CREATE/ALTER/DROP statements
- Retrieve value with the NEXT VALUE FOR clause

```sql
-- Define a sequence
CREATE SEQUENCE dbo.InvoiceSeq AS INT START WITH 1
INCREMENT BY 1;

-- Retrieve next available value from sequence
SELECT NEXT VALUE FOR dbo.InvoiceSeq;
```