CS108: Advanced Database

- Database Programming

Lecture 03:

The Foundation Statements of T-SQL

Expression

- Expression is a combination of symbols and operators that returns a single value
- An expression can be a single constant, variable, column, or scalar function
  - 10, 3.14, 'John Doe‘, '10/10/2010‘, CompanyName, GetDate()
- Columns, numbers, literals, functions connected by operators
  - 10*3+3, 'John' + 'Doe', Address + ', ' + City + ', ' + ZipCode
  - Quantity * SalePrice
  - 1.2 * ListPrice

Expressions used as Columns

- Expressions can be used as derived columns
- Examples

A text constant which will be the same for every record

  - SELECT CompanyName, 'Supplier' AS Type FROM Suppliers;
  - SELECT ProductName, UnitPrice*1.2 AS 'New Price'

    FROM Products;

String concatenation

  - SELECT FirstName+ ' ' +LastName FROM Employees;
  - SELECT UPPER(ProductName) FROM Products;
  - SELECT GETDATE();

A system function not related to any table

Predicates and Operators

- T-SQL has language elements in which predicates can be specified,
  - Query filters such as WHERE and HAVING, CHECK, …
- Predicates are logical expressions that evaluate to TRUE, FALSE, or UNKNOWN
- Predicates can be combined by using logical operators such as AND and OR

Predicates and Operators (Cont.)

- When multiple operators appear in the same expression, SQL Server evaluates them based on operator precedence rules.
  1. () (Parentheses)
  2. * (Multiplication), / (Division), % (Modulo)
  3. + (Positive), – (Negative), + (Addition), + (Concatenation), – (Subtraction)
  4. =, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
  5. NOT
  6. AND
  7. ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
  8. = (Assignment)

Example: Predicates and Operators

```
SELECT SalesOrderID, ProductID, OrderQty
FROM   Sales.SalesOrderDetail
WHERE
       ProductID = 700
   AND OrderQty IN (1, 3, 5)
   OR  ProductID = 900
   AND OrderQty IN (2, 4, 6);
```

```
SELECT SalesOrderID, ProductID, OrderQty
FROM   Sales.SalesOrderDetail
WHERE
        ( ProductID = 700
          AND OrderQty IN (1, 3, 5) )
   OR   ( ProductID = 900
          AND OrderQty IN (2, 4, 6) );
```

Operators

OPERATOR	EXAMPLE USAGE	EFFECT
=, >, <, >=, <=, <>, !=, !>, !<	<Column Name> = <Other Column Name> <Column Name> = ‘Bob’	These standard comparison operators work as they do in pretty much any programming language, with a couple of notable points: 1. What constitutes “greater than,” “less than,” and “equal to” can change depending on the collation order we have selected. 2. != and <> both mean “not equal.” !< and !> mean “not less than” and “not greater than,” respectively.
AND, OR, NOT	<Column1> = <Column2> AND <Column3> >= <Column 4> <Column1> != “MyLiteral” OR <Column2> = “MyOtherLiteral”	Standard Boolean logic. We can use these to combine multiple conditions into one WHERE clause. NOT is evaluated first, then AND, then OR. If we need to change the evaluation order, we can use parentheses. Note that XOR is not supported.

OPERATOR	EXAMPLE USAGE	EFFECT
BETWEEN	<Column1> BETWEEN 1 AND 5	Comparison is TRUE if the first value is between the second and third values, inclusive. It is the functional equivalent of A>=B AND A<=C. Any of the specified values can be column names, variables, or literals.
LIKE	<Column1> LIKE “ROM%” <Column1> LIKE ‘%!_%’ ESCAPE ‘!’	Uses the % and _ characters for wildcarding. • % indicates a value of any length can replace the % character. • _ indicates any one character can replace the _ character. • Enclosing characters in [ ] symbols indicates any single character within the [ ] is OK. ([a-c] means a, b, and c are OK. [ab] indicates a or b are OK.) • ^ operates as a NOT operator, indicating that the next character is to be excluded. • An escape character (ESCAPE keyword) help us to search for a character that is also used as a wildcard, (such as %, _, [, or]).

OPERATOR	EXAMPLE USAGE	EFFECT
IN	<Column1> IN (List of Numbers) <Column1> IN (“A”, “b”, “345”)	Returns TRUE if the value to the left of the IN keyword matches any of the values in the list provided after the IN keyword. This is frequently used in subqueries.
ALL, ANY, SOME	<column | expression> (comparison operator) <ANY|SOME> (subquery)	These return TRUE if any or all (depending on which we choose) values in a subquery meet the comparison operator’s (for example, <, >, =, >=) condition. • ALL indicates that the value must match all the values in the set. • ANY and SOME are functional equivalents and will evaluate to TRUE if the expression matches any value in the set.
EXISTS	EXISTS (subquery)	Returns TRUE if at least one row is returned by the subquery.
IS NULL	<Column1> IS NULL	It is not possible to test for NULL values with comparison operators, such as =, <, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

CASE Expressions

- A CASE expression is a scalar expression that returns a value based on conditional logic
- CASE is an expression and not a statement
  - It doesn’t let the control flow of activity or do something based on conditional logic
- CASE is a scalar expression, it is allowed wherever scalar expressions are allowed, such as
  - In the SELECT, WHERE, HAVING, and ORDER BY clauses and in CHECK constraints
- The CASE expression returns the highest precedence type from the set of types in the result expressions of the case expression

CASE Expressions - Simple Form

- The two forms of CASE expression are *simple* and *searched*
- The *simple* CASE form
    - Allows us to compare scalar expression with a list of possible values and return a value for the first match
    - If no value in the list is equal to the tested value, the CASE expression returns the value that appears in the ELSE
    - If a CASE expression doesn’t have an ELSE clause, it defaults to ELSE NULL

CASE Expressions - Searched Form

- The *searched* CASE form
  - More flexible because it allows us to specify predicates, or logical expressions
  - The searched CASE expression returns the value in the THEN clause that is associated with the first WHEN logical expression that evaluates to TRUE
  - If none of the WHEN expressions evaluates to TRUE, the CASE expression returns the value that appears in the ELSE clause (or NULL if an ELSE clause is not specified).

CASE Expressions

- Simple Form

```
CASE <test expression>
    WHEN <comparison expression1> THEN <return value1>
    WHEN <comparison expression2> THEN <return value2>
    WHEN <comparison expression3> THEN <return value3>
    WHEN <comparison expression4> THEN <return value4>
    [ELSE <value5>]
END
```

- Searched Form

```
CASE WHEN <test expression1> THEN <value1>
   [WHEN <test expression2> THEN <value2>]
   [WHEN <test expression3> THEN <value3>]
   [WHEN <test expression4> THEN <value4>]
   [ELSE <value5>]
END
```

Example: The Simple CASE Form

```
SELECT SalesOrderID, ProductID, SpecialOfferID,
   CASE SpecialOfferID
     WHEN 1  THEN 'No Discount'
     WHEN 2  THEN 'Volume Discount 11 to 14'
     WHEN 3  THEN 'Volume Discount 15 to 24'
     WHEN 4  THEN 'Volume Discount 25 to 40'
     WHEN 5  THEN 'Volume Discount 41 to 60'
     WHEN 6  THEN 'Volume Discount over 60'
     WHEN 7  THEN 'Mountain-100 Clearance Sale'
     WHEN 8  THEN 'Sport Helmet Discount-2002'
     WHEN 9  THEN 'Road-650 Overstock'
     WHEN 10 THEN 'Mountain Tire Sale'
     WHEN 11 THEN 'Sport Helmet Discount-2003'
     WHEN 12 THEN 'LL Road Frame Sale'
     WHEN 13 THEN 'Touring-3000 Promotion'
     WHEN 14 THEN 'Touring-1000 Promotion'
     WHEN 15 THEN 'Half-Price Pedal Sale'
     WHEN 16 THEN 'Mountain-500 Silver Clearance Sale'
     ELSE 'Unknown'
   END AS Description
FROM   Sales.SalesOrderDetail
```

Example: The Searched CASE Form

```
SELECT SalesOrderID, ProductID, SpecialOfferID,
   CASE
     WHEN SpecialOfferID = 1             THEN 'No Discount'
     WHEN SpecialOfferID BETWEEN 2 AND 6 THEN 'Volume Discount'
     WHEN SpecialOfferID = 8             THEN 'Mountain-100 Sale'
     WHEN SpecialOfferID IN (8, 11)      THEN 'Sport Discount'
     WHEN SpecialOfferID = 9             THEN 'Road-650 Overstock'
     WHEN SpecialOfferID = 10            THEN 'Mountain Tire Sale'
     WHEN SpecialOfferID = 12            THEN 'LL Road Frame Sale'
     WHEN SpecialOfferID IN (13, 14)     THEN 'Touring Promotion'
     WHEN SpecialOfferID = 15            THEN 'Half-Price Sale'
     WHEN SpecialOfferID = 16            THEN 'Mountain-500 Sale'
     ELSE 'Unknown'
   END AS Description
FROM   Sales.SalesOrderDetail
```

All-at-Once Operations

- SQL supports a concept called all-at-once operations, which mean all expressions are evaluated at the same point in time
- For example:

```
SELECT Color,
       YEAR(SellStartDate) AS OrderYear,
       OrderYear + 1       AS NextYear
FROM   Production.Product
```

  - The reference to the column alias OrderYear in the third expression in the SELECT list is invalid, even though the referencing expression appears “after” the one in which the alias is assigned.

Example: All-at-Once Operations

- Suppose we have a table called T1 with two integer columns called col1 and col2
- We want to return all rows for which col2/col1 is greater than 2.
- Because there may be rows in the table for which col1 is equal to zero, we need to ensure that the division doesn’t take place in those cases - otherwise, the query fails because of a divide-by-zero error.

Example: All-at-Once Operations

- So we write a query using the following format

```
SELECT col1, col2
FROM   T1
WHERE  col1 <> 0 AND col2/col1 > 2;
```

  - We might very well assume that SQL Server evaluates the expressions from left to right, and that if the expression col1 <> 0 evaluates to FALSE, SQL Server will short-circuit
  - SQL Server does support short circuits, because of the all-at-once operations concept in standard SQL;
  - This query might fail because of a divide-by-zero error.

Example: All-at-Once Operations

- We have several ways to avoid a failure. For example,

```
SELECT col1, col2
FROM   T1
WHERE  CASE
         WHEN col1 = 0      THEN 'no'
         WHEN col2/col1 > 2 THEN 'yes'
         ELSE 'no'
       END = 'yes';
```

- Or

```
SELECT col1, col2
FROM   T1
WHERE (col1 > 0 AND col2 > 2*col1)
   OR (col1 < 0 AND col2 < 2*col1);
```

String and Functions

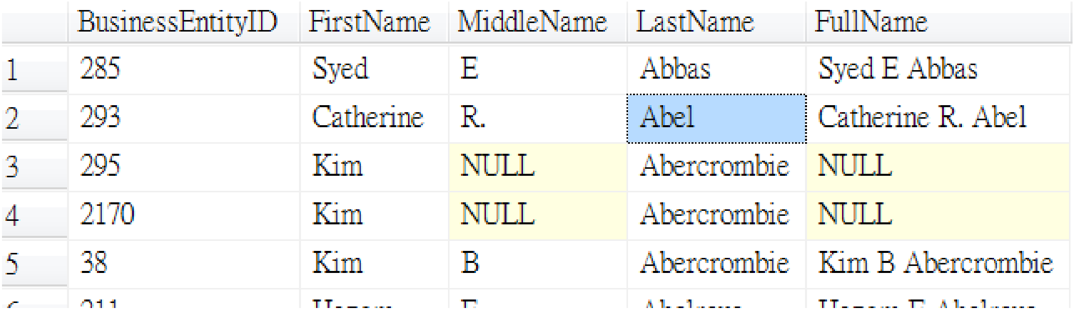
- SQL Server supports two kinds of character data types - regular and Unicode.
  - Regular data types include CHAR and VARCHAR, and
  - Unicode data types include NCHAR and NVARCHAR
- For string concatenation, T-SQL provides the + operator and the CONCAT function.
- For other operations on character strings, T-SQL provides several functions, including SUBSTRING, LEFT, RIGHT, LEN, DATALENGTH, REPLACE, LTRIM, FORMAT, etc.

String Concatenation

- String Concatenation (Plus Sign [+] Operator and CONCAT Function)
  - T-SQL provides the plus sign (+) operator and the CONCAT function (in SQL Server 2012) to concatenate strings.
  - Standard SQL dictates that a concatenation with a NULL should yield a NULL.
  - To treat a NULL as an empty string - or more accurately, to substitute a NULL with an empty string - we can use the COALESCE function.

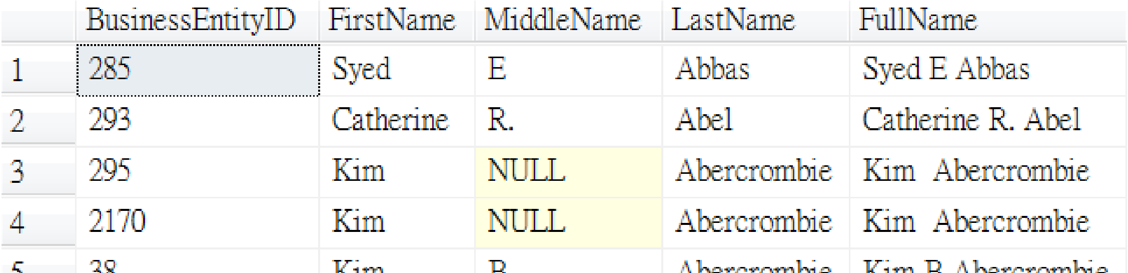
Example: String Concatenation

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName,
       FirstName + ' ' + MiddleName + ' ' + LastName AS
                                                FullName
FROM   Person.Person;
```



	BusinessEntityID	FirstName	MiddleName	LastName	FullName
1	285	Syed	E	Abbas	Syed E Abbas
2	293	Catherine	R.	Abel	Catherine R. Abel
3	295	Kim	NULL	Abercrombie	NULL
4	2170	Kim	NULL	Abercrombie	NULL
5	38	Kim	B	Abercrombie	Kim B Abercrombie
6	211	Hazem	E	Abolrous	Hazem E Abolrous

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName,
       FirstName + ' ' + COALESCE(MiddleName, '') + ' ' +
                                     LastName AS FullName
FROM   Person.Person;
```



	BusinessEntityID	FirstName	MiddleName	LastName	FullName
1	285	Syed	E	Abbas	Syed E Abbas
2	293	Catherine	R.	Abel	Catherine R. Abel
3	295	Kim	NULL	Abercrombie	Kim Abercrombie
4	2170	Kim	NULL	Abercrombie	Kim Abercrombie
5	38	Kim	B	Abercrombie	Kim B Abercrombie

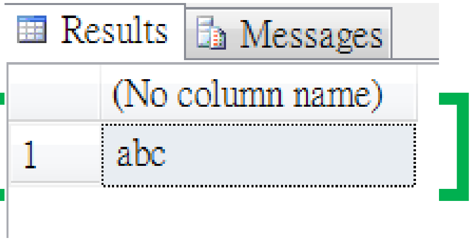
String Functions

- The SUBSTRING function extracts a substring from a string.

```
SUBSTRING(string, start, length)
```

- Example,

```
SELECT SUBSTRING('abcde', 1, 3);
```

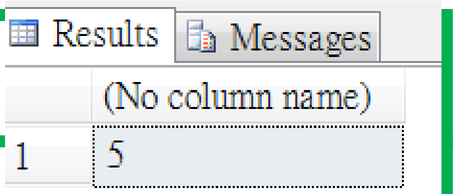


	(No column name)
1	abc

- The LEN and DATALENGTH functions return the number of characters in the input string
  - difference between LEN and DATALENGTH is that the former excludes trailing blanks but the latter doesn’t.

```
LEN(string)
```

```
SELECT LEN('abcde’);
```



	(No column name)
1	5

- The CHARINDEX function returns the position of the first occurrence of a substring within a string.

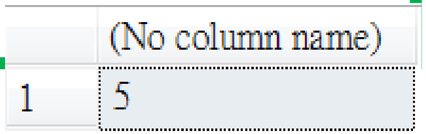
```
CHARINDEX(substring, string[, start_pos])
```

```sql
SELECT CHARINDEX(' ', 'Sio-Long Lo');
```

- The PATINDEX function returns the position of the first occurrence of a pattern within a string.
  - The argument pattern uses similar patterns used by the LIKE.

```
PATINDEX(pattern, string)
```

```sql
SELECT PATINDEX('%[0-9]%', 'abcd123efgh');
```

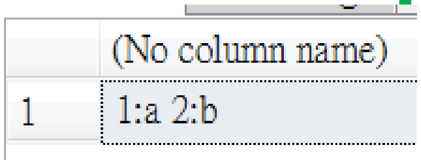


```
   (No column name)
1  5
```

  - The REPLACE function replaces a substring with another.

```
REPLACE(string, substring1, substring2)
```

```sql
SELECT REPLACE('1-a 2-b', '-', ':');
```



```
   (No column name)
1  1:a 2:b
```

Function	EXAMPLE USAGE	EFFECT
LEFT RIGHT	SELECT RIGHT('abcde', 3); --cde	• LEFT(string, n), RIGHT(string, n) • The LEFT and RIGHT functions are abbreviations of the SUBSTRING function, returning a requested number of characters from the left or right end of the input string.
REPLICATE	SELECT REPLICATE('abc', 3); --abcabcabc	• REPLICATE(string, n) • The REPLICATE function replicates a string a requested number of times.
STUFF	SELECT STUFF('xyz', 2, 1, 'abc'); --xabcz	• STUFF(string, pos, delete_length, insertstring) • The STUFF function allows you to remove a substring from a string and insert a new substring instead.

UPPER
LOWER
	SELECT UPPER(Sio-Long Lo');
	--SIO-LONG LO
	
	SELECT LOWER(‘Sio-Long Lo');
	--sio-long lo
		• UPPER(string), LOWER(string)
		• The UPPER and LOWER functions return the input string with all uppercase or lowercase characters, respectively.

Function	EXAMPLE USAGE	EFFECT
RTRIM LTRIM	SELECT RTRIM(LTRIM(' abc ')); --abc	• RTRIM(string), LTRIM(string) • The RTRIM and LTRIM functions return the input string with leading or trailing spaces removed.
FORMAT	SELECT FORMAT(1759, '000000000'); --000001759	• FORMAT(input , format_string, culture) • The FORMAT function allows you to format an input value as a character string based on a Microsoft .NET format string and an optional culture.

```
SELECT BusinessEntityID, LastName,
       LEN(LastName) - LEN(REPLACE(LastName, 'e', '')) AS [# of 'e']
FROM   Person.Person;
```

```
SELECT BusinessEntityID,
  RIGHT(REPLICATE('0', 9) + CAST(BusinessEntityID AS VARCHAR(10)), 10)
                                                          AS EntityID
FROM   Person.Person;
```

Date and Time Data

- SQL Server supported two date and time data types called DATETIME and SMALLDATETIME.
- The two data types differ in their storage requirements, their supported date range, and their accuracy.

Data Type	Storage (bytes)	Data Range	Accuracy	Recommended Entry Format and Example
DATETIME	8	January 1, 1753, through December 31, 9999	3 1/3 milliseconds	‘YYYYMMDD hh:mm:ss.nnn’ ‘20090212 12:30:15.123’
SMALLDATETIME	4	January 1, 1900, through June 6, 2079	1 minute	‘YYYYMMDD hh:mm’ ‘20090212 12:30’
DATE	3	January 1, 0001, through December 31, 9999	1 day	‘YYYY-MM-DD’ ‘2009-02-12’
TIME	3 to 5	N/A	100 nanoseconds	‘hh:mm:ss.nnnnnnn’ ‘12:30:15.1234567’

Function	EXAMPLE USAGE	EFFECT
DATEADD	SELECT DATEADD(year, 1, '20090212'); ----------------------- 2010-02-12 00:00:00.000	- DATEADD(part, n, dt_val) - The DATEADD function allows you to add a specified number of units of a specified date part to an input date and time value.
DATEDIFF	SELECT DATEDIFF(day, '20080212', '20090212'); --366	- DATEDIFF(part, dt_val1, dt_val2) - The DATEDIFF function returns the difference between two date and time values in terms of a specified date part.
DATEPART	SELECT DATEPART(month, '20090212'); --2	- DATEPART(part, dt_val) - The DATEPART function returns an integer representing a requested part of a date and time value.

YEAR
MONTH
DAY
	SELECT
	    DAY('20090212') AS D,
	    MONTH('20090212') AS M,
	    YEAR('20090212') AS Y;
	
	D M Y
	----------- ----------- -----------
	12 2 2009
		- YEAR(dt_val), MONTH(dt_val), DAY(dt_val)
		- The YEAR, MONTH, and DAY functions are abbreviations for the DATEPART function returning the integer representation of the year, month, and day parts of an input date and time value.

Date and Time Data

- SQL Server doesn’t provide the means to express a date and time literal; instead, it allows us to convert a literal - explicitly or implicitly - to a date and time data type.
- For example,

```
SELECT SalesOrderID, CustomerID, SalesPersonID, OrderDate
FROM   Sales.SalesOrderHeader
WHERE  OrderDate = '20120101';
```

- SQL Server recognizes the literal '20120101’ as a character string literal and converts it to the other’s type - DATETIME.

Date and Time Data

- It is important to note that some character string formats of date and time literals are language dependent.
- For example,

```sql
SET LANGUAGE British;
SELECT CAST('02/12/2007' AS DATETIME);
```

```
Changed language setting to British.
2007-12-02 00:00:00.000
```

```sql
SET LANGUAGE us_english;
SELECT CAST('02/12/2007' AS DATETIME);
```

```
Changed language setting to us_english.
2007-02-12 00:00:00.000
```

- Here, CAST(…) function explicitly convert string literal to DATETIME type.

Date and Time Data

- It is strongly recommended that we phrase our literals in a language-neutral manner.

Data Type	Format	Example
DATETIME	‘YYYYMMDD hh:mm:ss.nnn’ ‘YYYY-MM-DDThh:mm:ss.nnn’ ‘YYYYMMDD’	‘20090212 12:30:15.123’ ‘2009-02-12T12:30:15.123’ ‘20090212’
SMALLDATETIME	‘YYYYMMDD hh:mm’ ‘YYYY-MM-DDThh:mm’ ‘YYYYMMDD’	‘20090212 12:30’ ‘2009-02-12T12:30’ ‘20090212’
DATE	‘YYYYMMDD’ ‘YYYY-MM-DD’	‘20090212’ ‘2009-02-12’
TIME	‘hh:mm:ss.nnnnnnn’ ‘	‘12:30:15.1234567’
DATETIMEOFFSET	‘YYYYMMDD hh:mm:ss.nnnnnnn [+|-]hh:mm’ ‘YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-]hh:mm’ ‘YYYYMMDD’ ‘YYYY-MM-DD’	‘20090212 12:30:15.1234567 +02:00’ ‘2009-02-12 12:30:15.1234567 +02:00’ ‘20090212’ ‘2009-02-12’

Date and Time Data

- Language-neutral formats are always interpreted by SQL Server the same way and are not affected by language-related settings.

```
SET LANGUAGE British;
SELECT CAST('20070212' AS DATETIME);
```

```
Changed language setting to British.
2007-02-12 00:00:00.000
```

```
SET LANGUAGE us_english;
SELECT CAST('20070212' AS DATETIME);
```

```
Changed language setting to us_english.
2007-02-12 00:00:00.000
```

- Here, CAST(…) function explicitly convert string literal to DATETIME type.

Filtering Date Ranges

- When we need to filter a range of dates, such as a whole year or a whole month, it seems natural to use functions such as YEAR and MONTH.
- For example,

```
SELECT SalesOrderID, CustomerID, SalesPersonID, OrderDate
FROM   Sales.SalesOrderHeader
WHERE  YEAR(OrderDate) = 2012;
```

- To have the potential to use an index efficiently, we can

```
SELECT SalesOrderID, CustomerID, SalesPersonID, OrderDate
FROM   Sales.SalesOrderHeader
WHERE  OrderDate >= ‘20120101’ AND OrderDate < ‘20130101’
```

```
WHERE YEAR(OrderDate) = 2012 AND MONTH(OrderDate) = 2
OR
WHERE OrderDate >= ‘20120201’ AND OrderDate < ‘20120301’
```

The CAST Function

- The CAST, CONVERT and PARSE functions are used to convert an input value to some target type.
- If the conversion succeeds, the functions return the converted value; otherwise, they cause the query to fail.

```
▪ CAST(value AS datatype)
▪ TRY_CAST(value AS datatype)
▪ CONVERT(datatype, value [, style_number])
▪ TRY_CONVERT(datatype, value [, style_number])
▪ PARSE(value AS datatype [USING culture])
▪ TRY_PARSE(value AS datatype [USING culture])
```

```sql
SELECT CAST('20090212' AS DATE);
SELECT CAST('20090212' AS INT);
SELECT CAST(100 AS CHAR(10));
SELECT CAST(CAST('20090212' AS DATE) AS CHAR(25));
```

Data Modification

The INSERT Statement

- The INSERT INTO statement is used to insert new records in a table
- The full syntax for INSERT has several parts:

```
INSERT [TOP ( <expression> ) [PERCENT] ] [INTO] <tabular object>
  [(<column list>)]
  [ OUTPUT <output clause> ]
  { VALUES (<data values>) [,(<data values>)] } [, …n]
    | <table source>
    | EXEC <procedure>
    | DEFAULT VALUES
```

The INSERT Statement

- The more basic syntax for an INSERT looks like this:

```
INSERT [INTO] <table>
  [(<column list>)]
  VALUES (<data values>) [,(<data values>)] [, …n]
```

- The INTO keyword is optional
- An explicit column list is optional
  - If we don't provide an explicit column list, each value in the INSERT statement will be assumed to match up with a column in the same ordinal position of the table in order.
  - A value must be supplied for every column, in order, until we reach the last column that both does not accept NULLs and has no default.

The INSERT Statement

- The more basic syntax for an INSERT looks like this:

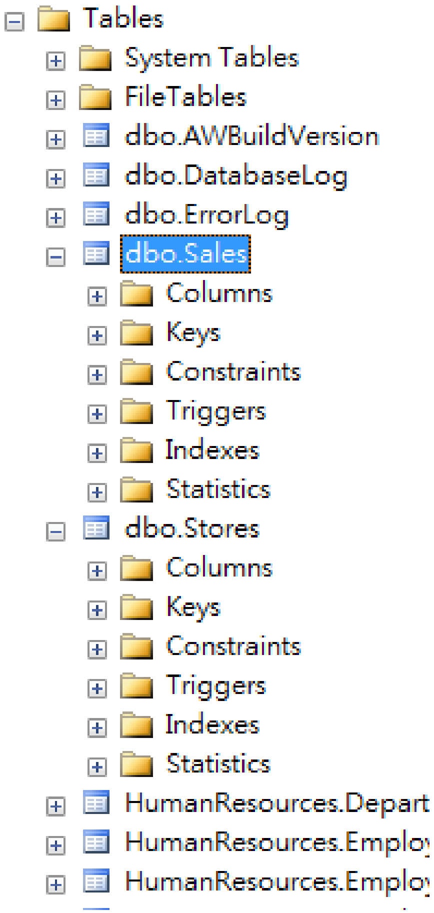
```
INSERT [INTO] <table>
  [(<column list>)]
  VALUES (<data values>) [,(<data values>)] [, …n]
```

- To insert the values:
  - Start with the VALUES keyword, and then follow that with a list of values separated by commas and enclosed in parentheses.
  - The number of items in the value list must exactly match the number of columns in the column list.
  - The data type of each value must match or be implicitly convertible to the type of the column with which it corresponds.

Example: The INSERT Statement

- Tables for our example:

```
CREATE TABLE Stores (
    StoreCode CHAR(4)     NOT NULL PRIMARY KEY,
    Name      VARCHAR(40) NOT NULL,
    Address   VARCHAR(40) NULL,
    City      VARCHAR(20) NOT NULL,
    State     CHAR(2)     NOT NULL,
    Zip       CHAR(5)     NOT NULL
);

CREATE TABLE Sales (
    OrderNumber VARCHAR(20) NOT NULL PRIMARY KEY,
    StoreCode   CHAR(4)     NOT NULL
        FOREIGN KEY REFERENCES Stores(StoreCode),
    OrderDate   DATE        NOT NULL,
    Quantity    INT         NOT NULL,
    Terms       VARCHAR(12) NOT NULL,
    TitleID     INT         NOT NULL
);
```



- Tables
  - System Tables
  - FileTables
  - dbo.AWBuildVersion
  - dbo.DatabaseLog
  - dbo.ErrorLog
  - dbo.Sales
    - Columns
    - Keys
    - Constraints
    - Triggers
    - Indexes
    - Statistics
  - dbo.Stores
    - Columns
    - Keys
    - Constraints
    - Triggers
    - Indexes
    - Statistics
  - HumanResources.Depart
  - HumanResources.Employ
  - HumanResources.Employ

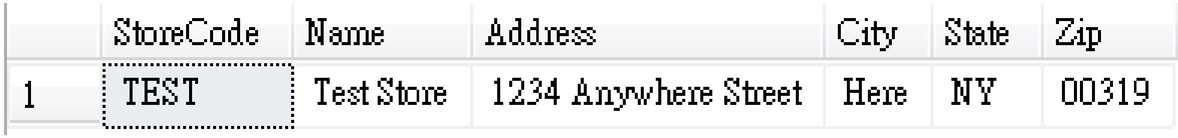
Example: The INSERT Statement

- To insert the data, we provide the value for every column:

```
INSERT INTO Stores
VALUES
(‘TEST’, ‘Test Store’, ‘1234 Anywhere Street’, ‘Here’, ‘NY’,
‘00319’);
```

- See what we just inserted

```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	Here	NY	00319

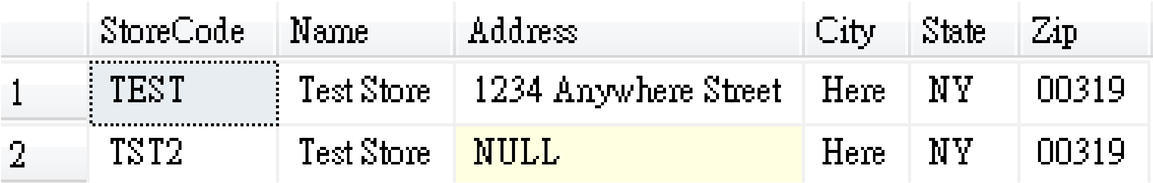
Example: The INSERT Statement

- We can omit the columns that accepts NULLs

```
INSERT INTO Stores
    (StoreCode, Name, City, State, Zip)
VALUES
    (‘TST2’, ‘Test Store’, ‘Here’, ‘NY’, ‘00319’);
```

- See what we just inserted

```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	Here	NY	00319
2	TST2	Test Store	NULL	Here	NY	00319

- A NULL was inserted for the column that we skipped

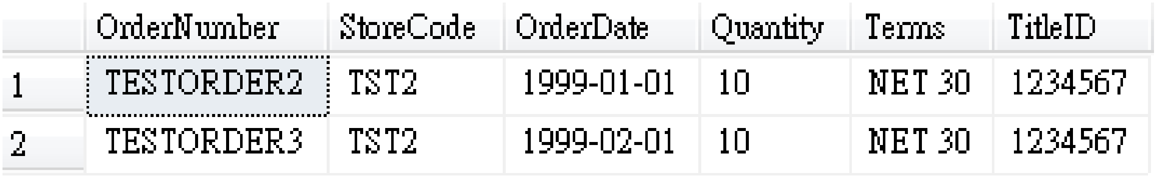
The INSERT Statement

- The column is not nullable, one of three conditions must exist, or we will receive an error and the INSERT will be rejected:
    - The column has been defined with a *default value*. A default is a constant value that is inserted if no other value is provided.
    - The column is defined to receive some form of *system-generated value*. The most common of these is an IDENTITY value. Other less common defaults may include SYSDATETIME() or a value retrieved from a SEQUENCE.
    - We supply a value for the column.

Multirow Inserts (for SQL Server)

- Starting with SQL Server 2008, we have the ability to insert multiple rows at one time. For example:

```
INSERT INTO Sales
    (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
    (‘TST2’, ‘TESTORDER2’, ‘01/01/1999’, 10, ‘NET 30’, 1234567),
    (‘TST2’, ‘TESTORDER3’, ‘02/01/1999’, 10, ‘NET 30’, 1234567);
```



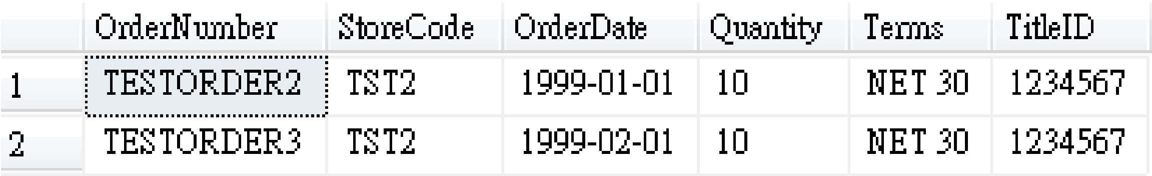
	OrderNumber	StoreCode	OrderDate	Quantity	Terms	TitleID
1	TESTORDER2	TST2	1999-01-01	10	NET 30	1234567
2	TESTORDER3	TST2	1999-02-01	10	NET 30	1234567

- This statement is processed as an atomic operation.

Multirow Inserts (Standard)

- We can use it in a standard way as a table value constructor to construct a derived table.
- For example:

```
SELECT *
FROM ( VALUES
          ('TST2', 'TESTORDER2', '01/01/1999', 10, 'NET 30', 1234567),
          ('TST2', 'TESTORDER2', '01/01/1999', 10, 'NET 30', 1234567)
      ) AS O(StoreCode, OrderNumber, OrderDate, Quantity, Terms,
                                                                  TitleID);
```



	OrderNumber	StoreCode	OrderDate	Quantity	Terms	TitleID
1	TESTORDER2	TST2	1999-01-01	10	NET 30	1234567
2	TESTORDER3	TST2	1999-02-01	10	NET 30	1234567

The INSERT SELECT Statement

- The INSERT INTO . . . SELECT statement can INSERT data from:
  - Another table in our database
  - A totally different database on the same server
  - A heterogeneous query from another SQL Server or other data
  - The same table
- The syntax for this statement comes from a combination of the two statements:

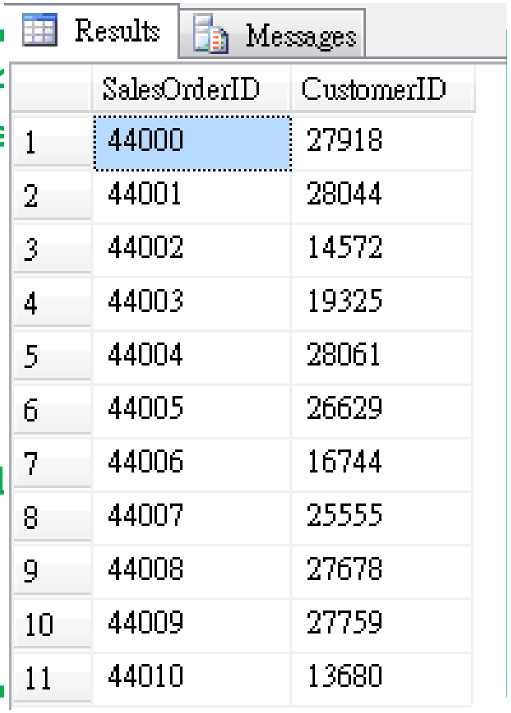
```
INSERT INTO <table name>
  [<column list>]
  <SELECT statement>
```

Example: INSERT INTO … SELECT

```
/* This particular table is actually a variable you are declaring
** on the fly. */
DECLARE @MyTable Table (
    SalesOrderID INT,
    CustomerID   CHAR(5)
);

/* Now that you have your table variable, you’re ready
** to populate it with data from your SELECT statement. */
INSERT INTO @MyTable
SELECT SalesOrderID, CustomerID
FROM   Sales.SalesOrderHeader
WHERE  SalesOrderID BETWEEN 44000 AND 44010;

-- Finally, make sure that the data was inserted like you think
SELECT *
FROM   @MyTable;
```

Example: INSERT INTO … SELECT

```
/* Now that you have your table variable, you're r
** to populate it with data from your SELECT state
INSERT INTO @MyTable
SELECT SalesOrderID, CustomerID
FROM   Sales.SalesOrderHeader
WHERE  SalesOrderID BETWEEN 44000 AND 44010;

-- Finally, make sure that the data was inserted l
SELECT *
FROM   @MyTable;
```



	SalesOrderID	CustomerID
1	44000	27918
2	44001	28044
3	44002	14572
4	44003	19325
5	44004	28061
6	44005	26629
7	44006	16744
8	44007	25555
9	44008	27678
10	44009	27759
11	44010	13680

- Note that if we try running a SELECT against @MyTable outside the current script, we're going to get an error.
- @MyTable is a declared variable, and it exists only as long as it remains in scope within the running batch. After that, it is automatically destroyed.

The DELETE Statement

- The DELETE statement is used to delete records in a table.
- There’s no column list, just a table name and (usually) a WHERE clause. The full version looks like this:

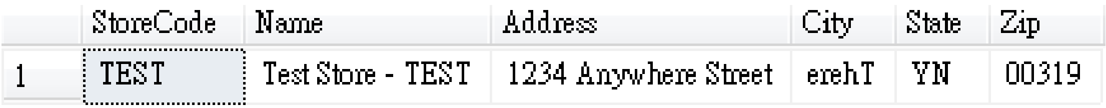
```
DELETE [TOP ( <expression> ) [PERCENT] ] [FROM] <tabular object>
    [ OUTPUT <output clause> ]
[FROM <table or join condition>]
[WHERE <search condition> | CURRENT OF [GLOBAL] <cursor name>]
```

- The basic syntax couldn’t be much easier:

```
DELETE <table name>
[WHERE <condition>]
```

Example: The DELETE Statement

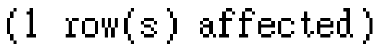
```
SELECT *
FROM   Stores
WHERE  StoreCode = 'TEST';
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store - TEST	1234 Anywhere Street	erehT	YN	00319

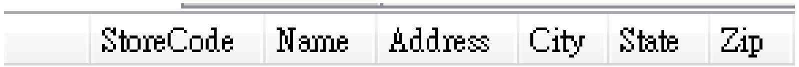
- We don’t need to provide a column list because we are deleting the entire row.

```
DELETE Stores
WHERE  StoreCode = ‘TEST’;
```



```
(1 row(s) affected)
```

```
SELECT *
FROM   Stores
WHERE  StoreCode = 'TEST';
```



	StoreCode	Name	Address	City	State	Zip

The TRUNCATE Statement

- T-SQL provides two statements for deleting rows from a table - DELETE and TRUNCATE.
- The TRUNCATE statement deletes all rows from a table. Unlike the DELETE statement, TRUNCATE has no filter.
- For example:

```
TRUNCATE TABLE T1;
```

- The advantage that TRUNCATE has over DELETE is that the former is minimally logged.

The UPDATE Statement

- The UPDATE statement, it updates existing data.
- The syntax as follows:

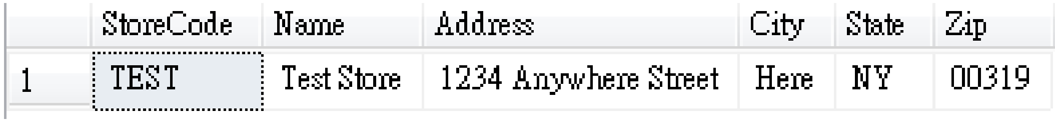
```
UPDATE [TOP ( <expression> ) [PERCENT] ] <tabular object>
   SET <column> = <value> [.WRITE(<expression>, <offset>, <length>)]
    [,<column> = <value> [.WRITE(<expression>, <offset>, <length>)]]
    [OUTPUT <output clause> ]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

- The more basic syntax:

```
UPDATE <table name>
SET    <column> = <value> [,<column> = <value>]
[FROM  <source table(s)>]
[WHERE <restrictive condition>]
```

Example: The UPDATE Statement

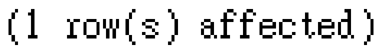
```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	Here	NY	00319

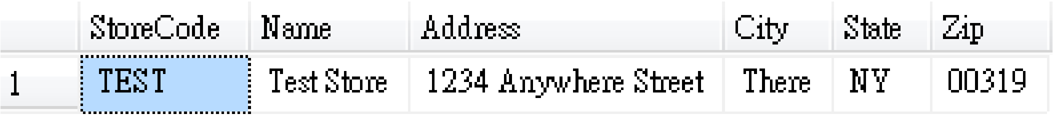
- Update the value in the City column:

```
UPDATE Stores
SET    City = 'There'
WHERE  StoreCode = 'TEST';
```



```
(1 row(s) affected)
```

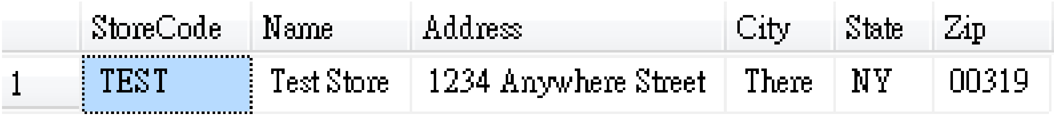
```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	There	NY	00319

Example: The UPDATE Statement

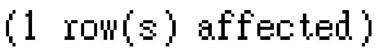
```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	There	NY	00319

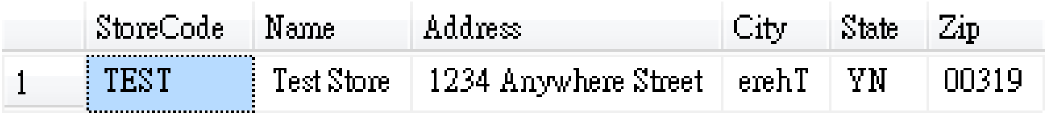
- We could update more than one column just by adding a comma and the additional column expression:

```
UPDATE Stores
SET    City = 'erehT', State = 'YN'
WHERE  StoreCode = 'TEST';
```



```
(1 row(s) affected)
```

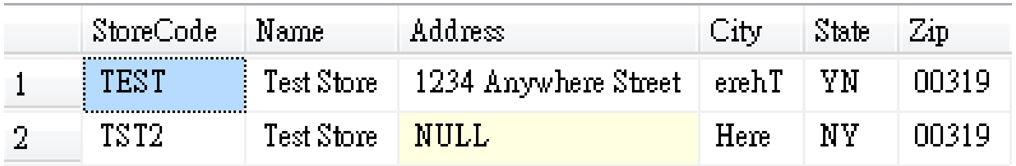
```
SELECT *
FROM   Stores
WHERE  StoreCode = ‘TEST’;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	erehT	YN	00319

Example: The UPDATE Statement

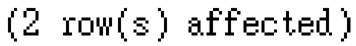
```
SELECT *
FROM   Stores;
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store	1234 Anywhere Street	erehT	YN	00319
2	TST2	Test Store	NULL	Here	NY	00319

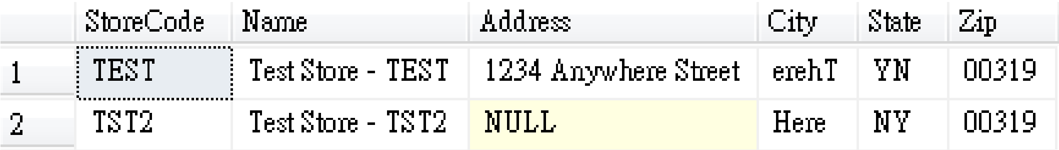
- We can use an expression for the SET clause instead of the explicit values:

```
UPDATE Stores
SET    Name = Name + ' - ' + StoreCode;
```



```
(2 row(s) affected)
```

```
SELECT *
FROM   Stores
```



	StoreCode	Name	Address	City	State	Zip
1	TEST	Test Store - TEST	1234 Anywhere Street	erehT	YN	00319
2	TST2	Test Store - TST2	NULL	Here	NY	00319

Summary

TOPIC	CONCEPT
SELECT	The most fundamental of all SQL DML (Data Manipulation Language) statements, SELECT is used to retrieve data from a table. We will commonly SELECT fields FROM tables WHERE some conditions are met.
GROUP BY	We may aggregate data in our SELECT statements, rather than simply return the rows exactly as they appear in the table, by using aggregators such as SUM, MAX, MIN, and AVG on the fields you want to aggregate and adding a GROUP BY clause to group by the other fields.
Filtering (WHERE) (HAVING)	Filtering with WHERE occurs before aggregation, and filtering with HAVING happens after. Filtering is done with Boolean tests.
DISTINCT	To remove duplicate rows from our results, we may use SELECT DISTINCT. Duplicates are checked across all fields returned.

Summary

TOPIC	CONCEPT
INSERT	To put data into a table, INSERT it. INSERT takes a list of the fields we want to insert into, the table we want to load the data into, and either a list of literal values using VALUES, or a SELECT statement that produces a compatible field list.
UPDATE	To alter data in a table, we use UPDATE table_name. Specify which values we want to change with SET field_name = value, and add a WHERE condition to limit the table rows updated.
DELETE	We can remove rows from our tables (permanently) with DELETE FROM table_name. Use a WHERE clause to limit the rows deleted.

Exercises

1. Write a query that outputs all of the columns and all of the rows from the Product table (in the Production schema) of the AdventureWorks database.
2. Modify the query in Exercise 1 so it filters down the result to just the products that have no ProductSubcategoryID. (HINT: There are 209, and you will need to be looking for NULL values.)
3. Add a new row into the Location (in the Production schema) table in the AdventureWorks database.
4. Remove the row you just added.

Exercises

5. Write a query against the Sales.SalesOrderHeader table that calculates row numbers for orders based on order date ordering (using the OrderDate as the tiebreaker) for each customer separately.

```
CustomerID  OrderDate               SalesPersonID Rownum
----------- ----------------------- ------------- --------------------
11000       2011-06-21 00:00:00.000 NULL          1
11000       2013-06-20 00:00:00.000 NULL          2
11000       2013-10-03 00:00:00.000 NULL          3
11001       2011-06-17 00:00:00.000 NULL          1
11001       2013-06-18 00:00:00.000 NULL          2
11001       2014-05-12 00:00:00.000 NULL          3
11002       2011-06-09 00:00:00.000 NULL          1
11002       2013-06-02 00:00:00.000 NULL          2
...
```

Exercises

6. Using the Person.Person table, figure out the SELECT statement that returns for each person the gender based on the title of courtesy. For ‘Ms. ‘ and ‘Mrs.’ return ‘Female’; for ‘Mr. ‘ return ‘Male’; and in all other cases return ‘Unknown’.

```
BusinessEntityID   FirstName     LastName      Title      Gender
----------------   ------------  ------------  ---------  -------
1                  Ken           Sánchez       NULL       Unknown
2                  Terri         Duffy         NULL       Unknown
3                  Roberto       Tamburello    NULL       Unknown
4                  Rob           Walters       NULL       Unknown
5                  Gail          Erickson      Ms.        Female
6                  Jossef        Goldberg      Mr.        Male
7                  Dylan         Miller        NULL       Unknown
8                  Diane         Margheim      NULL       Unknown
...
```

Production.Product

ProductID	int	Not null
Name	Name (user-defined type) nvarchar(50)	Not null
ProductNumber	nvarchar(25)	Not null
MakeFlag	Flag (user-defined type) bit	Not null
FinishedGoodsFlag	Flag (user-defined type) bit	Not null
Color	nvarchar(15)	null
SafetyStockLevel	smallint	Not null
ReorderPoint	smallint	Not null
StandardCost	money	Not null
ListPrice	money	Not null
Size	nvarchar(5)	null
SizeUnitMeasureCode	nchar(3)	null
WeightUnitMeasureCode	nchar(3)	null
Weight	decimal (8,2)	null
DaysToManufacture	int	Not null
ProductLine	nchar(2)	null
Class	nchar(2)	null
Style	nchar(2)	null
ProductSubcategoryID	smallint	null
ProductModelID	int	null
SellStartDate	datetime	Not null
SellEndDate	datetime	null
DiscontinuedDate	datetime	null
rowguid	uniqueidentifier ROWGUIDCOL	Not null
ModifiedDate	datetime	Not null

Sales.SalesOrderDetail

SalesOrderID	int	Not null	Primary key. Foreign key to SalesOrderHeader.SalesOrderID.
SalesOrderDetailID	int	Not null	Primary key. A sequential number used to ensure data uniqueness..
CarrierTrackingNumber	nvarchar(25)	Null	Shipment tracking number supplied by the shipper.
OrderQty	smallint	Not null	Quantity ordered per product.
ProductID	int	Not null	Product sold to customer. Foreign key to Product.ProductID.
SpecialOfferID	int	Not null	Promotional code. Foreign key to SpecialOffer.SpecialOfferID.
UnitPrice	money	Not null	Selling price of a single product.
UnitPriceDiscount	money	Not null	Discount amount.
LineTotal	Computed as OrderQty * UnitPrice	Not null	Per product subtotal.
rowguid	uniqueidentifier ROWGUIDCOL	Not null	ROWGUIDCOL number uniquely identifying the row. Used to support a merge replication sample.
ModifiedDate	datetime	Not null	Date and time the row was last updated.

Sales.SalesOrderHeader

SalesOrderID	int	Not null
RevisionNumber	tinyint	Not null
OrderDate	datetime	Not null
DueDate	datetime	Not null
ShipDate	datetime	Null
Status	tinyint	Not null
OnlineOrderFlag	Flag (user-defined type) bit	Not null
SalesOrderNumber	nvarchar(25)	Not null
PurchaseOrderNumber	OrderNumber (user-defined type) nvarchar(25)	Null
AccountNumber	AccountNumber (user-defined type) nvarchar(15)	Null
CustomerID	int	Not null
ContactID	int	Not null
SalesPersonID	int	Null
TerritoryID	int	Null
BillToAddressID	int	Not null
ShipToAddressID	int	Not null
ShipMethodID	int	Not null
CreditCardID	int	Null
CreditCardApprovalCode	varchar(15)	Null
CurrencyRateID	int	Null
SubTotal	money	Not null
TaxAmt	money	Not null
Freight	money	Not null
TotalDue	Computed as SubTotal + TaxAmt + Freight	Not null
Comment	nvarchar(128)	Null
rowguid	uniqueidentifier ROWGUIDCOL	Not null
ModifiedDate	datetime	Not null

Production.Location

LocationID	int	Not null	Primary key for Location records.
Name	Name (user-defined type) nvarchar(50)	Not null	Location description.
CostRate	smallmoney	Not null	Standard hourly cost of the manufacturing location.
Availability	decimal(8,2)	Not null	Work capacity (in hours) of the manufacturing location.
ModifiedDate	datetime	Not null	Date and time the row was last updated.

Person.Person

BusinessEntityID	int	Not null	Primary key for Person records.
PersonType	nchar(2)	Not null	Primary type of person: SC = Store Contact, IN = Individual (retail) customer, SP = Sales person, EM = Employee (non-sales), VC = Vendor contact, GC = General contact
NameStyle	NameStyle	Not null	0 = The data in FirstName and LastName are stored in western style (first name, last name) order. 1 = Eastern style (last name, first name) order.
Title	nvarchar(8)	Null	A courtesy title. For example, Mr. or Ms.
FirstName	Name	Not null	First name of the person.
MiddleName	Name	Null	Middle name or middle initial of the person.
LastName	Name	Not null	Last name of the person.
Suffix	nvarchar(10)	Null	Surname suffix. For example, Sr. or Jr.
EmailPromotion	int	Not null	0 = Contact does not wish to receive e-mail promotions, 1 = Contact does wish to receive e-mail promotions from AdventureWorks, 2 = Contact does wish to receive e-mail promotions from AdventureWorks and selected partners.
AdditionalContactInfo	xml	Null	Additional contact information about the person stored in xml format.
Demographics	xml	Null	Personal information such as hobbies, and income collected from online shoppers. Used for sales analysis.
rowguid	uniqueidentifier	Not null	ROWGUIDCOL number uniquely identifying the record. Used to support a merge replication sample.
ModifiedDate	datetime	Not null	Date and time the record was last updated.