T-SQL Fundamentals

Metadata

- Title: T-SQL Fundamentals, 3rd Edition
- Author: Itzik Ben-Gan
- Reference:

Notes

Create Database:

TSQLV4.sql 1 MB

Ch 2: Single-Table Queries

Elements of the SELECT Statement

T-SQL Syntax Order

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

T-SQL Logical Processing Order

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Clause: Syntactical component of a query (Ex: In the WHERE clause, you specify a predicate)
Phase: Logical manipulation taking place as part of a query (Ex: The WHERE phase returns rows for which the predicate evaluates to TRUE)

The FROM Clause

- Specify table names and table operators (if needed)
- Regular identifiers (table and column names) that comply with formatting rules do not need to be delimited
- Irregular identifies must be surrounded by " " or [ ]
- To return all rows from a table with no special manipulation, all you need is a query with a FROM clause with the specified table, and a SELECT clause with the specific attributes you want to return.

```
FROM Sales.Orders

OrderDetails -- regular
"Order Details" -- irregular
[Order Details] -- irregular
```

```
SELECT
    orderid,
    custid,
    empid,
    orderdate,
    freight
FROM Sales.Orders;
```

The WHERE Clause

- Predicates or logical expressions are specified in the WHERE clause
- Rows for which the logical expression evaluates to TRUE are returned, while FALSE and UNKOWN are discarded
- Filtering queries reduces network traffic created by returning all rows

```
SELECT
    orderid,
    custid,
    empid,
```

```
    orderdate,
    freight
FROM Sales.Orders
WHERE custid = 71;
```

The GROUP BY Clause

- Arranges rows returned by the previous logical processing phases into groups, determined by elements specified in the clause
- All phases subsequent to the GROUP BY phase (including HAVING, SELECT and ORDER BY) must operate on groups
- Each group is represented by a single row (scalar) in the final result
- Elements that are not in the GROUP BY clause must be inputted as aggregate functions in the SELECT statement
- *Note that all aggregate functions except for COUNT(*) ignore NULLs*

```
SELECT
  empid,
  YEAR(orderdate) AS orderyear,
  SUM(freight) AS totalfreight,
  COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

The HAVING Clause

- Group specific filter that filters elements in the GROUP BY clause
- Only groups for which the HAVING predicates evaluate to TRUE are returned, while FALSE and UNKNOWN are discarded
- Because the HAVING clause is processed after the GROUP BY clause, you can refer to aggregate functions in the logical expression

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
-- Returns employees who sold more than one order to customer 71, grouped by employee and
year.
```

The SELECT Clause

- The SELECT clause is where attributes (columns) are specified
- Expressions with no manipulations default to the source attribute name, while expressions with manipulations need to be aliased in order to return a column name
- Since the SELECT clause is processed after the FROM, WHERE, GROUP BY, and HAVING clauses, aliases assigned in the SELECT statement cannot be referenced in those clauses
- The DISTINCT clause can be used in the SELECT statement to remove duplicate rows
- SQL allows specifying an asterisk (*) in the SELECT list instead of specific attributes, returning all attributes in the specified table. This is considered bad programming practice in most cases, and individual attributes should be listed in the order in which you want them returned in the result set.

```
SELECT
    orderid,
    SUM(freight) AS totalfreight -- aliased column
FROM Sales.Orders
GROUP BY orderid;

SELECT DISTINCT
    empid,
    YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;

SELECT *
FROM Sales.Orders;
```

The ORDER BY Clause

- Used to sort the rows in the output for presentation purposes
- Without an ORDER BY clause, a query result has no guaranteed order
- Since the ORDER BY phase is processed after the SELECT phase where column aliases are defined, you can refer to the aliases

- Ascending order is the default, but you can specify sort order by adding ASC or DESC to the statemen (tt is considered best practice to specify ASC even though that is the default)
- You can specify elements in the ORDER BY clause that do not appear in the SELECT clause, unless a DISTINCT clause is specified.

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid ASC, orderyear ASC;
```

The TOP Filter

- T-SQL feature that limits the number or percentage of rows returned by the query
- It relies on two elements;
    - The number or percent of rows to return
    - A sort order specified in the ORDER BY clause (optional)
- When percent is used, the number of rows returned is calculated based on a percentage of the number of qualifying rows, rounded up
- If DISTINCT is specified in the SELECT statement, the TOP filter is evaluated after duplicate rows have been removed.
- The WITH TIES expression can be added to the TOP filter to return rows with the same sort value.

```
SELECT TOP (5)
    orderid,
    orderdate,
    custid,
    empid
FROM Sales.Orders
ORDER BY orderdate DESC; -- returns 5 rows

SELECT TOP (5) WITH TIES
    orderid,
    orderdate,
    custid,
    empid
```

```
FROM Sales.Orders
ORDER BY orderdate DESC; -- returns 8 rows due to duplicate orderdate results

SELECT TOP (100) *
FROM Sales.Orders; -- this query can be used to explore the table without pulling all rows

SELECT TOP (1) PERCENT
    orderid,
    orderdate,
    custid,
    empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The OFFSET-FETCH Filter

- The OFFSET-FETCH filter supports a skipping option, making it useful for ad-hoc paging purposes
- This filter is considered an extension to the ORDER BY clause.
- The OFFSET clause indicated how many rows to skip, and the FETCH clause indicates how many rows to return after the skipped rows
- The OFFSET-FETCH filter *must* have an ORDER BY clause
- The FETCH clause must be combined with the OFFSET clause. If you do not want to skip any rows, you can specify this with OFFSET 0 ROWS.
- OFFSET without FETCH is allowed, and will return all remaining rows in the result after the query skips the indicated number of rows in the OFFSET clause

```
SELECT
    orderid,
    orderdate,
    custid,
    empid
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Predicates and Operators

- Predicates are logical expressions that evaluate to TRUE, FALSE, or UNKNOWN
    - IN: Checks whether a value is equal to at least one of the elements in a specified set

    - BETWEEN: Checks whether a value is in a specified range, inclusive of the specified delimiters
    - LIKE: Checks whether a character string meets a specified pattern
- Predicates can be combined with logical operators such as AND, OR, NOT
- Standard comparison operators include: = , > , < , >= , <= , <> (not equal)
- Standard arithmetic operators include: + , - , * , ** , / , % (modulo)
- Standard null predicates include: IS NULL and IS NOT NULL
- When multiple operators appear in the same expression, SQL evaluates based on order of precedence (PEMDAS)
    - Parenthesis (Give you full control over precedence)
    - Multiplication, Division, Modulo
    - Positive, Negative, Addition, Concatenation, Subtraction
    - Comparison Operators
    - NOT
    - AND
    - BETWEEN, IN, LIKE, OR
    - Assignment (=)

```
- Predicates
SELECT
    orderid,
    empid,
    orderdate
FROM Sales.Orders
WHERE orderid IN(10248, 10249, 10250);

- Predicates and Logical Operators
SELECT
    orderid,
    empid,
    orderdate
FROM Sales.Orders
WHERE orderid BETWEEN 10300 AND 10310; -- inclusive

SELECT
    empid,
    firstname,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'D%'; -- N (NCHAR or NVARCHAR data types), % is a wild card, anything
follows
```

```
-- Comparison Operators
SELECT
    orderid,
    empid,
    orderdate
FROM Sales.Orders
WHERE orderdate >= '2016-01-01';

-- Predicates and Comparison Operators
SELECT
    orderid,
    empid,
    orderdate
FROM Sales.Orders
WHERE orderdate >= '2016-01-01' AND empid IN (1, 3, 5);

-- Arithmetic
SELECT
    orderid,
    productid,
    qty,
    unitprice,
    discount,
    qty * unitproce * (1 - discount) AS val
FROM Sales.orderDatails;

-- Precedence (AND has precedence over OR, despite the order query is written
SELECT
    orderid,
    custid,
    empid,
    orderdate
FROM Sales.Orders
WHERE
    custid = 1 AND
    empid IN(1, 3, 5) OR
    custid = 85 AND
    empid IN(2, 4, 6);
```

CASE Expressions

- Scalar expression that returns a value based on conditional logic
- Allowed in the SELECT, WHERE, HAVING, and ORDER BY clauses
- Two forms of CASE expressions are *simple* and *searched*

    - Use *simple* to compare one value or scalar expression with a list of possible values and return a value for the first match
    - Use *searched* when you need to specify predicates in the WHEN clause rather than being restricted to using equality comparisons
        - Returns the value in the THEN clause that is associated with the first WHEN predicate that evaluates to TRUE
- Functions that act as abbreviates CASE expressions include: ISNULL and COALESCE (standard)
    - ISNULL accepts two arguments as input and returns the first value that is not NULL, or NULL if both are NULL
    - COALESCE accepts two or more arguments as input and returns the first value that is not NULL, or NULL if both are NULL

```
-- Simple
SELECT
    productid,
    productname,
    categoryid,
    CASE categoryid
        WHEN 1 THEN 'Beverages'
        WHEN 2 THEN 'Condiments'
        WHEN 3 THEN 'Confections'
        WHEN 4 THEN 'Dairy Products'
        WHEN 5 THEN 'Grains/Cereals'
        WHEN 6 THEN 'Meat/Poultry'
        WHEN 7 THEN 'Produce'
        WHEN 8 THEN 'Seafood'
        ELSE 'Unknown Category' -- Optional, Defaults to ELSE IS NULL
    END AS categoryname
FROM Production.Products;

-- Searched (More Flexibility)
SELECT
    orderid,
    custid,
    val,
    CASE
        WHEN val < 1000.00 THEN 'Less than 1000'
        WHEN val BETWEEN 1000.00 and 3000.00 THEN 'Between 1000 and 3000'
        WHEN val > 3000.00 THEN 'More than 3000'
        ELSE 'Unknown' -- Optional, Defaults to ELSE IS NULL
    END AS valuecategory
FROM Sales.OrderValues;
```

```
-- These queries are for illustration purposes. In these specific cases, the category
names would be stored in a seperate table that can be joined.

-- ISNULL
SELECT ISNULL('Hello', 'World') -- Returns 'Hello'
SELECT ISNULL(NULL, 'World') -- Returns 'World'

-- COALESCE
SELECT COALESCE('Hello', NULL, 'World', NULL, NULL) -- Returns 'Hello'
SELECT COALESCE(NULL, NULL, 'World', NULL, 'World') -- Returns 'World'
```

Character Data Types

- SQL Server supports two kinds of character data types:
  - Regular data types include CHAR and VARCHAR (1 byte per character)
  - Unicode data types include NCHAR and NVARCHAR (4 bytes per character)
    - When expressing a Unicode character literal, the character *N* (for National) is prefixed before the string
  - Data types without the VAR element in its name (CHAR, NCHAR) have a fixed length
  - Data types with the VAR element in its name (VARCHAR, NVARCHAR) have a variable length
    - SQL Server will use as much storage space as needed in the row
- Single quotes are used to delimit character strings
- Double quotes (or square brackets) are used to delimit irregular identifiers such as table or column names

Functions

- Common T-SQL functions;

```
CONCAT(StringValue1, StringValue2, StringValueN)

SUBSTRING(String, Start, Length)

LEFT(String, N), RIGHT(String, N) -- N is the number of characters to extract from the
left or right end of the supplied string

LEN(String)

DATALENGTH(Expression)
```

```
REPLACE(String, Substring1, Substring2) -- Replaces all occurences of of Substring1 with
Substring2

REPLICATE(String, N) -- N is the number of times the string is replicated

UPPER(String), LOWER(String)

LTRIM(String), RTRIM(String) -- Removes leading or trailing spaces

FORMAT(Value, Format, Culture)
```

The LIKE Predicate

- Checks whether a character string matches a specified pattern, supported by wildcard characters
    - The percent wildcard represents a string of any size, including an empty string
    - The underscore wildcard represents a single character
    - The [List] wildcard represents a single character that must be one of the specified characters in the list
    - The [Range] wildcard represents a single character that must be within the specified range
    - The [^List or Range] wildcard represents a single character that is NOT within the specified list or range

```
-- Percent Wildcard
SELECT
    empid,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'D%'; -- Returns any lastname starts with D, with any length.

-- Underscore Wildcard
SELECT
    empid,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'_e%'; -- Returns any lastname with an e for the second character,
with any length after the wildcard.

-- [List] Wildcard
SELECT
    empid,
```

```
    lastname
FROM HR.Employees
WHERE lastname LIKE N'[ABC]%'; -- Returns any lastname with A, B, or C as a first
character, with any length.

-- [Range] Wildcarad
SELECT
    empid,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'[A-E]%'; -- Returns any lastname with A, B, C, D, or E as a first
character, with any length.

-- [^ List or Range] Wildcard
SELECT
    empid,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'[^A-E]%'; -- Returns any last name that does NOT start with A, B, C,
D, or E, with and length.
```

The ESCAPE Character

- If you want to search for a character that is also used as a wildcard (such as %, _ , [ ]), you can use as escape character
- Specify a character that would not appear in the data as the escape character in front of the character you are searching for
- Specify the keyword ESCAPE followed by the escape character immediately after the patters

```
SELECT
    empid,
    lastname
FROM HR.Employees
WHERE lastname LIKE N'%!_%' ESCAPE '!'; -- Returns any last name with an underscore in the
name (zero results)
```

Date & Time Data Types

- T-SQL supports six data and time data types
- DATETIME and SMALLDATETIME are legacy types
- DATE, TIME, DATETIME2, and DATETIMEOFFSET are later additions

- These types differ in storage requirements, supported date range, and precision
-

Data type	Storage (bytes)	Date 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' '20160212 12:30:15.123'
SMALLDATETIME	4	January 1, 1900, through June 6, 2079	1 minute	''YYYYMMDD hh:mm' '20160212 12:30'
DATE	3	January 1, 0001, through December 31, 9999	1 day	'YYYY-MM-DD' '2016-02-12'
TIME	3 to 5	N/A	100 nanoseconds	'hh:mm:ss.nnnnnnn' '12:30:15.1234567'
DATETIME2	6 to 8	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn' '2016-02-12 12:30:15.1234567'
DATETIMEOFFSET	8 to 10	January 1, 0001, through December 31, 9999	100 nanoseconds	'YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-] hh:mm' '2016-02-12 12:30:15.1234567 +02:00'

-

Data type	Accuracy	Recommended entry format and example
DATETIME	'YYYYMMDD hh:mm:ss.nnn' 'YYYY-MM-DDThh:mm:ss.nnn' 'YYYYMMDD'	'20160212 12:30:15.123' '2016-02-12T12:30:15.123' '20160212'
SMALLDATETIME	'YYYYMMDD hh:mm' 'YYYY-MM-DDThh:mm' 'YYYYMMDD'	'20160212 12:30' '2016-02-12T12:30' '20160212'
DATE	'YYYYMMDD' 'YYYY-MM-DD'	'20160212' '2016-02-12'
DATETIME2	'YYYYMMDD hh:mm:ss.nnnnnnn' 'YYYY-MM-DD hh:mm:ss.nnnnnnn' 'YYYY-MM-DDThh:mm:ss.nnnnnnn' 'YYYYMMDD' 'YYYY-MM-DD'	'20160212 12:30:15.1234567' '2016-02-12 12:30:15.1234567' '2016-02-12T12:30:15.1234567' '20160212' '2016-02-12'
DATETIMEOFFSET	'YYYYMMDD hh:mm:ss.nnnnnnn [+|-]hh:mm' 'YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-]hh:mm' 'YYYYMMDD' 'YYYY-MM-DD'	'20160212 12:30:15.1234567 +02:00' '2016-02-12 12:30:15.1234567 +02:00' '20160212' '2016-02-12'
TIME	'hh:mm:ss.nnnnnnn'	'12:30:15.1234567'

Converting to Date

- Used to express dates in language-dependent formats
- The CONVERT function converts a character-string literal to a requested data type, with a specified styling number
- https://docs.microsoft.com/en-us/sql/t-sql/functions/cast-and-convert-transact-sql?view=sql-server-ver15

```
SELECT CONVERT(DATE, '02/12/2016', 101);
-- Returns 2016-02-12

SELECT CONVERT(DATE, '02/12/2016', 103);
-- Returns 2016-12-02
```

Filtering Date Ranges

- Filtering date ranges such as a whole year or month can be done with functions such as YEAR and MONTH
    - This eliminates the possibility of using efficient indexing
- Alternatively, range filtering can be used to maintain index possibilities

```
-- Date Filter Functions
SELECT
    orderid,
    custid,
    empid,
    orderdate
FROM Sales.Orders
WHERE YEAR(orderdate) = 2015 AND MONTH(orderdate) = 01; -- Returns all orders from Jan 2016

-- Date Filter Range
SELECT
    orderid,
    custid,
    empid,
    orderdate
FROM Sales.Orders
WHERE orderdate >= '2016-01-01' AND orderdate < '2016-02-01'; -- Returns all orders from
Jan 2016
```

Current Date & Time Functions

Function	Return type	Description
GETDATE	*DATETIME*	Current date and time
CURRENT_TIMESTAMP	*DATETIME*	Same as GETDATE but ANSI SQL–compliant
GETUTCDATE	*DATETIME*	Current date and time in UTC
SYSDATETIME	*DATETIME2*	Current date and time
SYSUTCDATETIME	*DATETIME2*	Current date and time in UTC
SYSDATETIMEOFFSET	*DATETIMEOFFSET*	Current date and time, including the offset from UTC

```
SELECT
  GETDATE()           AS [GETDATE],
  CURRENT_TIMESTAMP   AS [CURRENT_TIMESTAMP],
  GETUTCDATE()        AS [GETUTCDATE],
```

```
  SYSDATETIME()       AS [SYSDATETIME],
  SYSUTCDATETIME()    AS [SYSUTCDATETIME],
  SYSDATETIMEOFFSET() AS [SYSDATETIMEOFFSET];
```

CAST, CONVERT, and PARSE Functions with TRY Counterparts

- Used to convert an input value to some type target type, returning a converted value if the conversion succeed and an error if it does not
- Each function has a TRY counterpart, that returns a NULL instead of an error in the case where a conversion fails
- CAST is standard, while CONVERT and PARSE are not
  - PARSE is more expensive than CONVERT so it will not be covered
- It is recommended to use CAST unless style numbers or culture are needed
- Syntax:
  - 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*])

```
-- CAST
SELECT CAST('20160212' AS DATE); -- Returns 2016-02-12

SELECT CAST(SYSDATETIME() AS DATE); -- Returns YYYY-MM-DD

SELECT CAST(SYSDATETIME() AS TIME); -- Returns hh:mm:ss:nnnnnnn

-- CONVERT
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP, 112); -- Returns a string 'YYYYMMDD'

SELECT CONVERT(CHAR(12), CURRENT_TIMESTAMP, 114); -- Returns a string 'hh:mm:ss:nnnnnnn'
```

The DATEADD Function

- Adds a specified number of units of a specified date part to an input date and time value
- Syntax: DATEADD(*part, n, value*)
- Valid values for part include;

    - YEAR
    - QUARTER
    - MONTH
    - DAYOFYEAR
    - DAY
    - WEEK
    - WEEKDAY
    - HOUR
    - MINUTE
    - SECOND
    - MILLISECOND
    - MICROSECOND
    - NANOSECOND
- The return type for a date and time input is the same as the input type
- If the function is given a string literal as input, the output is DATETIME

```
SELECT DATEADD(YEAR, 1, '20160212'); -- Returns 2017-02-12 00:00:00.000

SELECT DATEADD(MONTH, 3, '2016-02-01'); -- Returns 2016-05-01 00:00:00.000

SELECT DATEADD(MONTH, -1, CAST('2016-02-1' AS DATE)); Returns 2016-01-01
```

The DATEDIFF and DATEDIFF_BIG Functions

- Return the difference between two date and time values in terms of a specified date part
- DATEDIFF returns a INT value (4-byte integer), while DATEDIFF_BIG returns a BIGINT value(8-byte integer)
- Syntax:
    - DATEDIFF(*part*, *dt_val1*, *dt_val2*)
    - DATEDIFF_BIG(*part*, *dt_val1*, *dt_val2*)
- Valid values for part are the same for the DATEADD function above

```
SELECT DATEDIFF(DAY, '2016-01-01', '2016-12-31'); -- Returns 365

SELECT DATEDIFF(MONTH, '2016-01-01', '2016-12-31'); -- Returns 11
```

The DATEPART Function

- Returns an integer representing a requested part of a date and time value
- Syntax: DATEPART(*part, dt_val*)
- Valid values for part are the same for the DATEADD functions above, but also include:
  - TZoffset
  - ISO_WEEK

```
SELECT DATEPART(MONTH, '2016-01-01'); -- Returns 1 for January

SELECT DATEPART(YEAR, '2016-01-01'); -- Returns 2016
```

The YEAR, MONTH, and DAY Functions

- Abbreviations of the DATEPART function, returning an integer representation of the year, month, and day parts of an input date and time value
- Syntax:
  - YEAR(*dt_val*)
  - MONTH(*dt_val*)
  - DAY(*dt_val*)

```
SELECT
  DAY('20160212') AS TheDay,
  MONTH('20160212') AS TheMonth,
  YEAR('20160212') AS TheYear;
```

The DATENAME Function

- Returns a character string representing a part of a date and time value
- Similar to DATEPART with the same *part* input options, returning a name where relevant (if the part requested does not have a name, the function returns a numeric value)
- Syntax: DATENAME(*dt_val, part*)

```
SELECT DATENAME(MONTH, '2016-01-01'); -- Returns January

SELECT DATENAME(YEAR, '2016-01-01'); -- Returns 2016 (year does not have a name)
```

THE ISDATE Function

- Accepts a character string as an argument, and returns 1 if it is convertible to a date and time data type, or 0 if it isn't
- Syntax: ISDATE(*string*)

```
SELECT ISDATE('20160212'); -- Returns 1 (Yes)

SELECT ISDATE('20160230'); -- Returns 0 (No)

SELECT ISDATE('TEST'); -- Returns 0 (No)
```

The FROMPARTS Function

- Accepts integer inputs representing parts of a date and time value and constructs a value of the requested type from those parts
- Syntax:
  - DATEFROMPARTS (*year*, *month*, *day*)
  - DATETIME2FROMPARTS (*year*, *month*, *day*, *hour*, *minute*, *seconds*, *fractions*, *precision*)
  - DATETIMEFROMPARTS (*year*, *month*, *day*, *hour*, *minute*, *seconds*, *milliseconds*)
  - DATETIMEOFFSETFROMPARTS (*year*, *month*, *day*, *hour*, *minute*, *seconds*, *fractions*, *hour_offset*, *minute_offset*, *precision*)
  - SMALLDATETIMEFROMPARTS (*year*, *month*, *day*, *hour*, *minute*)
  - TIMEFROMPARTS (*hour*, *minute*, *seconds*, *fractions*, *precision*)

```
SELECT
  DATEFROMPARTS(2016, 02, 12), -- Returns 2016-02-12
  DATETIME2FROMPARTS(2016, 02, 12, 13, 30, 5, 1, 7), -- Returns 2016-02-12 13:30:05.0000001
  DATETIMEFROMPARTS(2016, 02, 12, 13, 30, 5, 997), -- Returns 2016-02-12 13:30:05.997
  DATETIMEOFFSETFROMPARTS(2016, 02, 12, 13, 30, 5, 1, -8, 0, 7), -- Returns 2016-02-12
13:30:05.0000001 -08:00
  SMALLDATETIMEFROMPARTS(2016, 02, 12, 13, 30), -- Returns 2016-02-12 13:30:00
  TIMEFROMPARTS(13, 30, 5, 1, 7); -- Returns 13:30:05.0000001
```

The EOMONTH Function

- Accepts an input date and time value and returns the respective end-of-month date as a DATE typed value
- Supports an optional second argument indicating how many months to add or subtract (negative)
- Syntax: EOMONTH(*input* [, *months_to_add*])

```
SELECT EOMONTH('2016-01-01'); -- Returns 2016-01-31

SELECT EOMONTH('2016-01-01', 3); -- Returns 2016-04-30

SELECT
    orderid,
    orderdate,
    custid,
    empid
FROM Sales.Orders
WHERE orderdate = EOMONTH(orderdate); -- Returns all orders placed on the last day of the
month
```

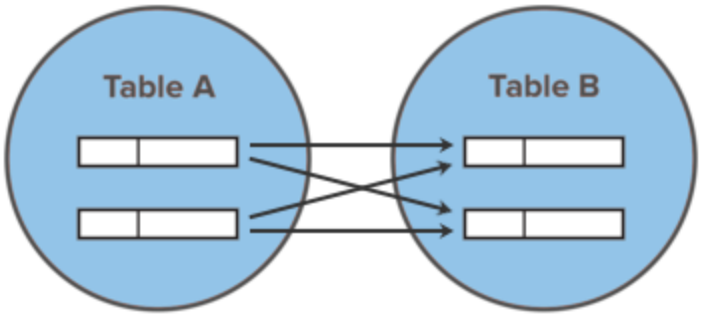
Ch 3: Joins

Introduction

- The FROM clause is the first to be logically processed
- Within the FROM clause, table operators operate on input tables
  - JOIN, APPLY, PIVOT, UNPIVOT *(This chapter covers JOIN)*
- Each table operator acts on input tables, applies a set of logical processing phases, and returns a table result
- JOIN operates on two input tables and has three fundamental types
  - CROSS JOIN
  - INNER JOIN
  - LEFT/RIGHT OUTER JOIN
- Assigning table aliases is a best practice
  - Assigning an alias to the source table requires column names from that table to be prefixed with the same alias

CROSS JOIN

- Implements one logical query processing phase - a Cartesian product
- Each row from one input table is matched with all rows from another input table
  - If Table A has *m* rows, and Table B has *n* rows, a cross join produces *m x n* rows

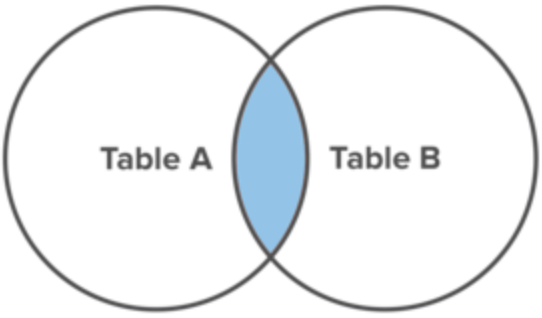


SQL CROSS JOIN

```
SELECT
    C.custid,
    E.empid
FROM Sales.Customers AS C
    CROSS JOIN HR.Employees AS E; -- Results in 819 rows (91 rows in Customers, 9 rows in
Employees)
```

INNER JOIN

- Implements two logical query processing phases - a Cartesian product, a filter based on a specified predicate (join condition)
  - The join condition is an ON statement and specifies which columns to match
- The INNER keyword is options because inner joins are default, but it is a best practice to include it
- The result set is a table that matches each record from Table A with all related records from Table B (that have a matching join condition)
  - Any record from Table A without a related record in Table B is discarded



SQL INNER JOIN

```
SELECT
    E.empid,
    E.firstname,
    E.lastname,
    O.orderid
FROM HR.Employees AS E
    INNER JOIN Sales.Orders AS O
        ON E.empid = O.empid; -- Results in 819 rows (91 rows in Customers, 9 rows in
Employees). All rows have a match, so none are discarded.
```

Multi-Join Queries

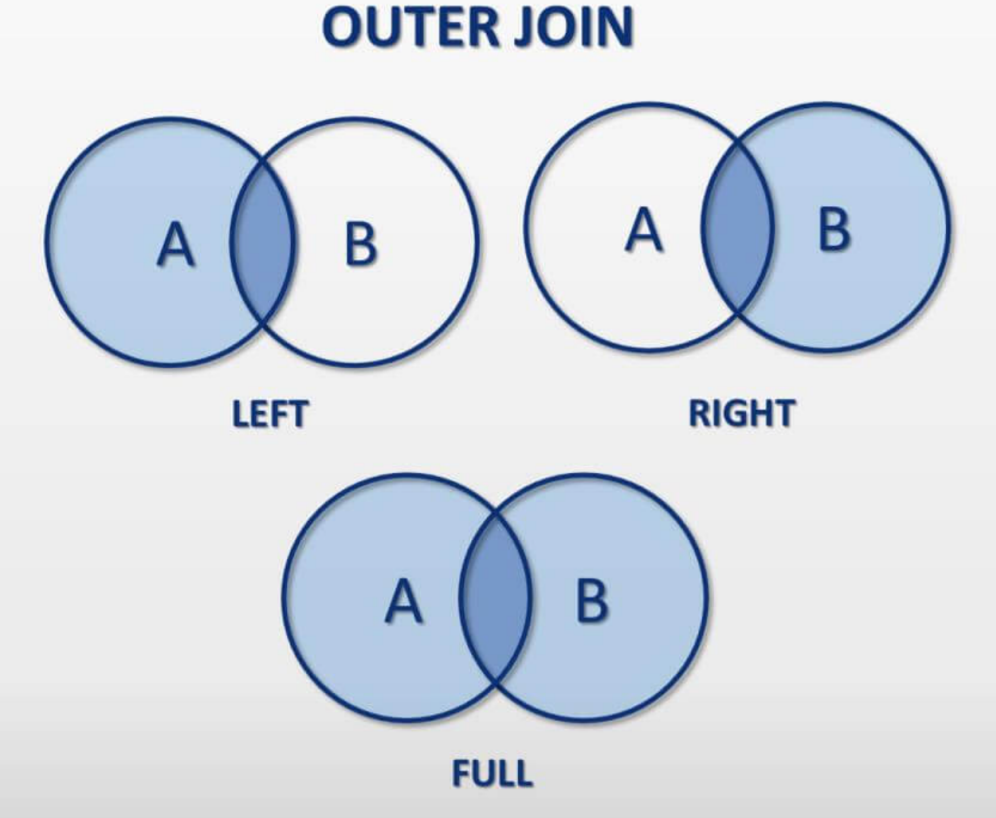
- A join table operator only operates on two tables, but a single query can have multiple joins
- Multiple join table operators are processed from left to right
    - Result table of the first table operator is treated as the left input to the second table operator and so on
    - The first join operates on two base tables, all other subsequent joins use the result of the preceding join as their left input

```
SELECT
    c.custid,
    C.companyname,
    O.orderid,
    OD.productid,
    OD.qty
FROM Sales.Customers AS C
    INNER JOIN Sales.Orders AS O -- Results in all records from Customers with a matching
record from Orders
        ON C.custid = O.custid
    INNER JOIN Sales.OrderDetails AS OD -- Results in all records from the preceding set
with a matching record from OrderDetails
        ON O.orderid = OD.orderid;
```

OUTER JOIN (LEFT & RIGHT)

- Outer joins apply three logical processing phases - a Cartesian product, a filter based on a specified predicate (join condition), adds specified *outer* rows
- Tables are marked as preserved by prefixing OUTER JOIN with LEFT, RIGHT, or FULL
    - LEFT OUTER JOIN: Rows from the left table are preserved

    - RIGHT OUTER JOIN: Rows from the right table are preserved
    - FULL OUTER JOIN: Rows from both left and right tables are preserved
- The third phase of the outer join identifies rows from the preserved table that does not have matching rows from the non-preserved table, and adds these rows back to the result table with NULLs as a placeholder for attributes from the non-preserved side
    - Rows that are are added back from an outer join were discarded in the filter phase
    - ***It might help to think of the result of an outer join as having two kinds of rows with respect to the preserved side—inner rows and outer rows. Inner rows are rows that have matches on the other side based on the ON predicate, and outer rows are rows that don't. An inner join returns only inner rows, whereas an outer join returns both inner and outer rows.***
- You can specify final filtering predicates in the WHERE clause to return only outer rows
    - Be careful when filtering the non-preserved side
-




```
SELECT
```

```
    C.custid,
    C.companyname,
    O.orderid
FROM Sales.Customers AS C
    LEFT OUTER JOIN Sales.Orders AS O
        ON C.custid = O.custid; -- Customers that have not placed an order will be in the
result table with NULLs for orderid

SELECT
    C.custid,
    C.companyname,
    O.orderid
FROM Sales.Customers AS C
    LEFT OUTER JOIN Sales.Orders AS O
        ON C.custid = O.custid
WHERE O.orderid IS NULL; -- Returns outer rows only. Customers without a matching order.
```

Ch 4: Subqueries

- SQL supports writing queries within queries, or nesting queries
- The outermost query is returned to the caller and is known as the outer query
- The innermost query's result is used by the outer query and is known as the inner query
- This eliminates the need for separate steps in your solution, like declaring variables
- Subqueries can be self-contained or correlated
- Subqueries can return a single value, multiple values, or a whole table result (reviewed in Ch 5)

Self-Contained Subqueries

- Subqueries that are independent of the tables in the outer query
- Logically, subqueries are evaluated once before the outer query is evaluated
    - The outer query uses the results of the subquery

Self-Contained Scaler Subquery Examples

- Returns a single value (query will fail otherwise)

    - ```
      Msg 512, Level 16, State 1, Line 40
      Subquery returned more than 1 value. This is not permitted when the
      subquery follows =, !=, <,
      <= , >, >= or when the subquery is used as an expression.
      ```
    - If a scalar subquery (inner query) returns no value, the empty result is converted to NULL,
      - The outer query then returns an empty set
- Can appear anywhere in the outer query where a single-valued expression can appear
    - WHERE (Using the = comparison operator to return a scalar result)
    - SELECT

```
-- Query the Orders table and return information about the order with the maximum orderID
in the table

-- Using a variable (less efficient)
DECLARE @maxid AS INT = (SELECT MAX(orderid) FROM Sales.Orders);

SELECT
    orderid,
    orderdate,
    empid,
    custid
FROM Sales.Orders
WHERE orderid = @maxid; -- Filters query where orderid equals the declared variable (max
orderid)

-- Using a self-contained scalar subquery (more efficient)
SELECT
    orderid,
    orderdate,
    empid,
    custid
FROM Sales.Orders
WHERE orderid = (SELECT MAX(O.orderid)
                 FROM Sales.Orders AS O)

-- Non-scalar subquery (fails)
SELECT orderid
FROM Sales.Orders
WHERE empid = (SELECT E.empid
               FROM HR.Employees AS E
               Where E.lastname LIKE N'D%'); -- Multiple employees with a lastname
starting with 'D'
```
```

```
-- NULL value subquery (returns empty set)
SELECT orderid
FROM Sales.Orders
WHERE empid =
    (SELECT E.empid
    FROM HR.Employees AS E
    WHERE E.lastname LIKE N'A%'); -- No lastnames that begin with A
```

These last two examples are for illustration purposes only. Because an equality operator is used in the WHERE clause, these are considered scalar. But because these subqueries can potentially return more than 1 result, the equality predicate is incorrectly used. If the subquery happens to return 0 or 1 value, it runs. Otherwise, it will fail. These queries should be handled with an IN predicate in a multivalued subquery (below).

Self-Contained Multivalued Subquery Examples

- Returns multiple values in a single column
- The IN predicate can be used to operate on multi-valued subqueries
    - <Scalar_Expression> IN <Multivalued Subquery>
    - The predicate evaluates to true if the *scalar expression* is equal to any of the values returned by the subquery
    - You can use the NOT IN predicate to negate the IN predicate

```
SELECT orderid
FROM Sales.Orders
WHERE empid IN
    (SELECT E.empid
    FROM HR.Employees AS E
    WHERE E.lastname LIKE N'D%');

-- Negating the IN predicate
SELECT
    custid,
    companyname
FROM Sales.Customers
WHERE custid NOT IN
    (SELECT O.custid
    FROM Sales.Orders AS O); -- Returns customers with no orders (NOT IN) (Best practice
is to qualify subquery to exclude NULLs, will be reviewed later)

-- The first query problem can technically be solved by joining Orders and Employees
SELECT O.orderid
FROM HR.Employees AS E
```

```
INNER JOIN Sales.Orders AS O
    ON E.empid = O.empid
WHERE E.lastname LIKE N'D%');
```

Query problems can sometimes be solved multiple ways. Keep an eye on performance when choosing which to use.

Correlated Subqueries

- Refer to attributes from the tables that appear in the outer query
- Subquery is dependent on the outer query and cannot be invoked independently

```
SELECT
    custid,
    orderid,
    orderdate,
    empid
FROM Sales.Orders AS O1
WHERE O1.orderid =
    (SELECT MAX(O2.orderid)
    FROM Sales.Orders AS O2
    WHERE O2.custid = O1.custid); -- OrderID equals the value returned by the subquery

SELECT
    orderid,
    custid,
    val,
    CAST(100.0 * val / (
        SELECT SUM(O2.val)
        FROM Sales.OrderValues AS O2
        WHERE O2.custid = O1.custid) AS NUMERIC(5, 2)
    ) AS pct
FROM Sales.OrderValues AS O1
ORDER BY custid, orderid; -- Percentage of the current order value out of the customer
total
```

The EXISTS Predicate

- Predicate accepts a subquery as input and returns TRUE if the subquery returns any rows and FALSE otherwise
- You can negate the EXISTS predicate with the NOT operator
- EXISTS uses two-valued logic (TRUE, FALSE) and not three-valued logic (TRUE, FALSE, UNKOWN)

```
-- Return customers from Spain who did place orders
SELECT
    custid,
    companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
    AND EXISTS (
        Select * FROM Sales.Orders AS O
        WHERE O.custid = C.custid
    ); -- The outer query filters customers from Spain for whom the EXISTS predicate
returns true (current customer has related orders in the Orders table)

-- Returns customers from Spain who did NOT place orders
SELECT
    custid,
    companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
    AND NOT EXISTS (
        Select * FROM Sales.Orders AS O
        WHERE O.custid = C.custid
    ); -- The outer query filters customers from Spain for whom the EXISTS predicate
reutnrs true (current customer does not have related orders in the Orders table)
```

Beyond the Fundamentals of Subqueries

- Returning previous or next values (more efficient with LAG and LEAD window functions)
    - Complication arises since result sets have no order, so an expression needs to be written that obtains maximum (or minimum) value that is smaller (or larger) than the current value
- Returning a running total (window functions that accomplish this task will be reviewed later)
    - You use a correlated subquery to sum the total of previous years with the current year

```
-- Returning previous orders
SELECT
    orderid,
    orderdate,
    empid,
    custid,
    (SELECT MAX(O2.orderid)
```

```
    FROM Sales.Orders AS O2
    WHERE O2.orderid < O1.orderid) AS prevorderid
FROM Sales.Orders AS O1;

-- Returning Next Orders
SELECT
    orderid,
    orderdate,
    empid,
    custid,
    (SELECT MIN(O2.orderid)
    FROM Sales.Orders AS O2
    WHERE O2.orderid > O1.orderid) AS nextorderid
FROM Sales.Orders AS O1;

-- Running Aggregate
SELECT
    orderyear,
    qty,
    (SELECT SUM(O2.qty)
    FROM Sales.OrderTotalsByYear AS O2
    WHERE O2.orderyear <= O1.orderyear) AS runqty -- Sums all previous year with current
year (running total)
FROM Sales.OrderTotalsByYear AS O1
ORDER BY orderyear;
```

Dealing with Misbehaving Subqueries

- NULL trouble
- Substitution errors in subquery column names

Ch 5: Table Expressions

- Table expressions represents a valid relational table (temporary)
- T-SQL supports four types of table expressions;
  - Derived tables
  - Common table expressions (CTEs)
  - Views
  - Inline table-valued functions (inline TVFs)
- Table expressions are not physically materialized anywhere, they are virtual

Derived Tables

- Also known as table subqueries, defined in the FROM clause of an outer query
- As soon as the outer query is finished, the derived table is gone
- The derived table is defined within parenthesis, followed by an AS clause and derived table name
- Valid table subqueries (inner queries) must meet three requirments
    - Order is not guaranteed (an ORDEY BY clause is not allowed)
    - All columns must have names, aliases must be used when necessary
    - All column names must be unique

```
-- Simple example of basic syntax, a derived table is not technically needed because the
outer query is not performing any manipulation
SELECT *
FROM (
    SELECT custid, companyname
    FROM Sales.Customers
    WHERE country = N'USA'
) AS USACustomers
```

Assigning Column Aliases

- Aliasing columns in the inner query allows you to reference the alias in the SELECT and GROUP BY clauses of the outer query (usually not the case)
    - Since the FROM clause is processed first, the inner query is processed, making the aliases available for reference
- Table expressions are used for logical reasons, not performance
    - Allowing the use of aliases in the outer query improves readability
    - When debugging, running the inner query isolated produces a result set with column aliases (instead of no_column)
- Table expressions have neither a positive or negative impact on performance
- Syntax: <expression> AS <alias>

```
-- Table expression with column aliases
SELECT
    orderyear,
    COUNT (DISTINCT custid) AS numcusts
FROM (
    SELECT
        YEAR (orderdate) AS orderyear,
```

```
        custid
    FROM Sales.Orders
) AS D
GROUP BY orderyear;

-- Standard query that produces same result
SELECT
    YEAR (orderdate) AS orderyear,
    COUNT (DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR (orderdate);
```

Using Arguments

- Queries that define a derived table can refer to pre-defined arguments such as declared variables, functions, and stored procedures

```
-- Table expression that references a declared variable
DECLARE @empid AS INT = 3

SELECT
    orderyear,
    COUNT (DISTINCT custid) as numcusts
FROM (
    SELECT
        YEAR (orderdate) AS orderyear,
        custid
    FROM Sales.Orders
    WHERE empid = @empid
) AS D
GROUP BY orderyear; -- Returns distinct customers per year whose orders were handled by
the input empoyee (@empid)
```

Multiple References

- Multiple instances of the same query can be written but can't reference each other
- This leads to lengthy code that is prone to errors, but the example illustrated below presents an interesting use case

```
-- LEFT OUTER JOIN Table Expression
SELECT
    CurrentYear.orderyear,
    CurrentYear.numcusts AS curnumcusts,
    PreviousYear.numcusts AS prvnumcusts,
```

```
    CurrentYear.numcusts - PreviousYear.numcusts AS growth
FROM (
    SELECT YEAR (orderdate) AS orderyear, COUNT (DISTINCT custid) AS numcusts
    FROM Sales.Orders
    GROUP BY YEAR (orderdate)
) AS CurrentYear
LEFT OUTER JOIN (
    SELECT YEAR (orderdate) AS orderyear, COUNT (DISTINCT custid) AS numcusts
    FROM Sales.Orders
    GROUP BY YEAR (orderdate)
) AS PreviousYear
ON CurrentYear.orderyear = PreviousYear.orderyear + 1; -- ensures each year from the first
derived table matches the previous year of the second

-- Output
orderyear    curnumcusts    prvnumcusts    growth
2014         67             NULL           NULL
2015         86             67             19
2016         81             86             -5
```

Common Table Expressions (CTEs)

- Another standard form of table expressions, similar to derived tables
- CTEs are defined using a WITH statement and follows the general form below
- The inner query must follow all requirements for table expressions
- As soon as the outer query finishes, the CTE goes out of scope (temporary table)
- To avoid ambiguity, when the WITH clause is used to define a CTE, the preceding statement in the same batch must be terminated with a semicolon
- *The inner query of a CTE (first query) creates a temporary table that the outer query can query*
    - *The temporary table is NOT stored in memory*

```
-- CTE general form
WITH <CTE Name> [(<target column list)]
AS <alias>
(
    <inner query defining CTE>
)
<outer query against CTE>

-- CTE simple example
WITH USACusts AS (
    SELECT custid, companyname
    FROM Sales.Customers
```

```
    WHERE country = N'USA'
) -- creates cte returning all customers from the US (think of this as a temp table)
SELECT *
FROM USACusts; -- queries all rows and columns from the cte (temp table)
```

Assigning Column Aliases in CTEs

- CTEs also support inline and external column aliasing
  - Internal: Specify <expression> AS <column_alias>
  - External: Specify the target column list in parenthesis immediately after the CTE name
  - *Both produce the same result, the difference is readability*

```
-- Internal Aliasing
WITH CustTemp AS
(
    SELECT YEAR (orderdate) AS orderyear, custid
    FROM Sales.Orders
)
SELECT
    orderyear,
    COUNT (DISTINCT custid) AS numcusts
FROM CustTemp
GROUP BY orderyear;

-- External Aliasing
WITH CustTemp (orderyear, custid) AS
(
    SELECT YEAR (orderdate) AS orderyear, custid
    FROM Sales.Orders
)
SELECT
    orderyear,
    COUNT (DISTINCT custid) AS numcusts
FROM CustTemp
GROUP BY orderyear;
```

Using Arguments in CTEs

- CTEs also support the use of arguments in the inner query (similar to derived tables)

```
-- CTE Arguments
DECLARE @empid AS INT = 3; -- preceding statement MUST be ended with semicolon
```

```
WITH CustTemp AS
(
    SELECT YEAR (orderdate) AS orderyear, custid
    FROM Sales.Orders
    WHERE empid = @empid
)
SELECT
    orderyear,
    COUNT (DISTINCT custid) AS numcusts
FROM CustTemp
GROUP BY orderyear; -- returns number of customers from declared empoyee id
```

Defining Multiple CTEs

- An advantage of CTEs when compared to derived tables (table expressions) is that each CTE can refer to previously defined CTEs, and the outer query can reference all CTEs
- This improves readability and maintainability as opposed to nesting derived tables
- *In the example below, the first two queries are CTEs (inner queries, aka temporary tables), the last query is the outer query that references the above temporary tables*

```
-- Multiple CTEs (seperated by whitespace for readability)

WITH CustTemp_1 AS
(
    SELECT YEAR (orderdate) AS orderyear, custid
    FROM Sales.Orders
), -- CTEs must be seperated by commas

CustTemp_2 AS
(
    SELECT orderyear, COUNT (DISTINCT custid) AS numcusts
    FROM CustTemp_1
    GROUP BY orderyear
)

SELECT
    orderyear,
    numcusts
FROM CustTemp_2
WHERE numcusts > 70;
```

Multiple References in CTEs

- Since the CTE exists when you write the outer query, you can reference multiple instances of the temporary table in a table operator (such as a join clause)
- This improves readability and allows you to define a temporary table only once

```
WITH YearlyCount AS (
    SELECT
        YEAR (orderdate) AS orderyear,
        COUNT (DISTINCT custid) AS numcusts
    FROM Sales.Orders
    GROUP BY Year (orderdate)
)
SELECT
    cur.orderyear,
    cur.numcusts AS curnumcusts,
    prev.numcusts AS prevnumcusts,
    cur.numcusts - prev.numcusts AS growth
FROM YearlyCount AS cur
LEFT OUTER JOIN YearlyCount AS prev
    ON cur.orderyear = prev.orderyear + 1 -- join condition states current year is the
previous year + 1
```

Views

- Derived tables and CTEs have a single-statement scope (they are not reusable)
- Views and inline table-valued functions (TVFs) are table expressions that are stored as permanent objects in the database
- As with derived tables and CTEs, you can use external column aliasing instead of inline column aliasing if you wish
- After a view is created, you can query it the same way you query other tables
- Because a view is a database object, access permissions can be managed similar to tables, and direct access to the underlying objects can be denied while granting access to the view itself
- Best practice is to explicitly list column names desired int he CREATE VIEW statement
    - You can always use the ALTER VIEW statement to revise if needed

```
DROP VIEW IF EXISTS Sales.USACusts; -- Prevents duplication of exisiting view
GO

CREATE VIEW Sales.USACusts AS (
```

```
    SELECT
        custid,
        companyname,
        contactname,
        contacttitle,
        address,
        city,
        region,
        postalcode,
        country,
        phone,
        fax
    FROM Sales.Customers
    WHERE country = N'USA'
);
GO

-- Querying a view
SELECT *
FROM Sales.USACusts;

-- Altering a view to remove the fax number
ALTER VIEW Sales.USACusts AS (
    SELECT
        custid,
        companyname,
        contactname,
        contacttitle,
        address,
        city,
        region,
        postalcode,
        country,
        phone
    FROM Sales.Customers
    WHERE country = N'USA'
);
GO
```

Views and the Order By Clause

- The query used to define a view must meet the same inner query requirements as derived tables and CTEs
    - No guaranteed order

        - Exception when using TOP, OFFSET-FETCH, or FOR XML options but this it outside the scope of this book
    - All columns have names that are distinct
- If a presentation is required, you can specify an order by using the ORDER BY clause in a query against the view (outer query)

View Options: ENCRYPTION

- The ENCRYPTION option is available when you create or alter a view, stored procedure, trigger, or user-defined function (UDF)
- SQL Server will then store the object definition in an obfuscated format
- *Whenever you alter a view, you need to repeat the objects specified in the original view definition if you wish to keep them*

```
-- Getting an object definition
SELECT OBJECT_DEFINITION (OBJECT_ID ('Sales.USACusts'));

-- Altering a view to add encryption
ALTER VIEW Sales.USACusts WITH ENCRYPTION AS (
    SELECT
        custid,
        companyname,
        contactname,
        contacttitle,
        address,
        city,
        region,
        postalcode,
        country,
        phone,
        fax
    FROM Sales.Customers
    WHERE country = N'USA'
);
GO

-- Now the following query will return NULL
SELECT OBJECT_DEFINITION (OBJECT_ID ('Sales.USACusts'));
```

View Options: SCHEMABINDING

- The SCHEMABINDING options is available when you create or alter a view or UDF

- It binds the schema of referenced objects and columns to the schema of the referencing object
- This indicates that referenced objects (views) cannot be dropped, and referenced columns cannot be dropped or altered
- This presents potential issues when you try to query a view and the referenced objects or columns do not exists
- The object definition must meet certain requirements to support the SCHEMABINDING option (these are best practices anyway)
    - Query is not allowed to use * in the SELECT clause, all column names must be explicitly listed
    - Schema qualified two-part names must be used when referring to objects (Schema.TableName)

```
-- Altering view to add schema binding
ALTER VIEW Sales.USACusts WITH SCHEMABINDING AS (
    SELECT
        custid,
        companyname,
        contactname,
        contacttitle,
        address,
        city,
        region,
        postalcode,
        country,
        phone,
        fax
    FROM Sales.Customers
    WHERE country = N'USA'
);
GO

-- Any attempts to drop a view, or drop/alter a column will result in an error
```

View Options: CHECK

- Without the check option, records can be inserted to a view that contradict the filtering defined in the view
- This will add the record to the source table, but will not be available in the view since it does not meet the filter logic

    - In this case, it would make more sense to insert the record directly into the source table
- Adding the check option prevents modifications that conflict with the view's filters

```
-- Altering view to add check option
ALTER VIEW Sales.USACusts WITH SCHEMABINDING AS (
    SELECT
        custid,
        companyname,
        contactname,
        contacttitle,
        address,
        city,
        region,
        postalcode,
        country,
        phone,
        fax
    FROM Sales.Customers
    WHERE country = N'USA'
) WITH CHECK OPTION;
GO
```

Inline (Table-Valued Functions: TVFs)

- Inline TVFs are reusable table expressions that support input parameters
    - Similar to views and can be considered parameterized views
- When called, input parameters are specified in the parenthesis following the function's name
- It's a best practice to provide an alias when calling an inline TVF

```
-- Create a TVF
USE TSQLV4;
DROP FUNCTION IF EXISTS dbo.GetCustOrders;
GO
CREATE FUNCTION dbo.GetCustOrders
    (@custid AS INT) RETURNS TABLE -- @custid is the input parameter (customer id)
AS
RETURN
    SELECT
        orderid,
        custid,
        empid,
        orderdate,
        requireddate,
```

```
        shippeddate,
        shipperid,
        freight,
        shipname,
        shipaddress,
        shipcity,
        shipregion,
        shippostalcode,
        shipcountry
    FROM Sales.Orders
    WHERE custid = @custid
GO

-- Query a TVF
SELECT
    orderid,
    custid
FROM dbo.GetCustOrders(1) AS O; -- returns all orders from customer id 1
```

Ch 6: Set Operators

- Set operators combine rows from two queries, whether a row is returned depends on the outcome of the comparison and operator used
  - The operator is applied to the results of the two queries
- UNION, UNION ALL, INTERSECT, EXCEPT
  - Two 'flavors' of each operators - DISTINCT (default) and ALL
  - Distinct remove duplicates, returning a set (ALL returns a multi-set)
  - *DISTINCT is implicit, you can't specify it explicitly*
- The individual queries cannot have an ORDER BY clause, but you can optionally add one to the result set of the operator
- The individual queries must produce results with the same number of columns and data types (if not of same type, the lower data type in precedence must be convertible to the higher data type)
  - You can explicitly apply this conversion in the individual queries and that is preferred
- The names of the columns in the results are determined by the first query, you should assign aliases there if needed

- Set operators allow you to circumvent writing special treatment for NULLs, as when NULLs are compared across both queries they return a TRUE and will be in the result set

```
-- Basic Syntax
Input Query 1
< Set Operator >
Input Query 2
[ORDER BY];
```

The UNION Operator

- Unifies the results of two input queries, if a row appears in any of the input sets it will appear in the result of the UNION operator
- The individual queries do not have to include all attributes of the source tables
- T-SQL supports both UNION ALL and UNION (implied DISTINCT)
- Use UNION if duplicates are possible in the unified result and you do not need to return them, otherwise use UNION AL
- If duplicates cannot exist, both UNION and UNION ALL will return the same result
    - In this case, UNION ALL should be used to prevent a performance loss related to checking for duplicates

The UNION ALL Operator

- Unifies the two input query results without attempting to remove duplicates (the result is a multiset)
- If Query 1 returns n rows and query 2 returns m rows, UNION ALL returns m + n rows

```
-- UNION ALL
SELECT
    country,
    region,
    city
FROM HR.Employees

UNION ALL

SELECT
    country,
    region,
    city
FROM Sales.Customers
```

The UNION (DISTINCT) Operator

- Unifies the results of two input queries and eliminates duplicates (the result is a set)

```
-- UNION (DISTINCT)
SELECT
    country,
    region,
    city
FROM HR.Employees

UNION -- DISTINCT is implied

SELECT
    country,
    region,
    city
FROM Sales.Customers
```

The INTERSECT (DISTINCT) Operator

- Returns only the rows that are common in both input query results, removing duplicates
- The operator results will include a NULL record
    - If instead you used an inner join or correlated subquery, you would need add special treatment for NULLs

```
-- INTERSECT (DISTINCT)
SELECT
    country,
    region,
    city
FROM HR.Employees

INTERSECT -- DISTINCT is implied

SELECT
    country,
    region,
    city
FROM Sales.Customers
```

The EXCEPT (DISTINCT) Operator

- Returns only rows that appear in the first query but not the second, removing duplicates
- The order of the two queries matters, since this is similar to a left outer join on the first query
- There are alternatives to the EXCEPT operator such as outer joins and the NOT EXISTS predicate
    - These require you to be explicit about comparisons and treatment for NULLs

```
-- EXCEPT (DISTINCT)

-- Returns distinct locations that are employee locations but not customer locations
SELECT
    country,
    region,
    city
FROM HR.Employees

EXCEPT -- DISTINCT is implied

SELECT
    country,
    region,
    city
FROM Sales.Customers


-- Returns distinct locations that are cutomer locations but not employee locations
SELECT
    country,
    region,
    city
FROM Sales.Customers

EXCEPT -- DISTINCT is implied

SELECT
    country,
    region,
    city
FROM HR.Employees
```

Precedence

- In a query that contains multiple set operators;
  - The INTERSECT operator precedes UNION and EXCEPT
  - UNION and EXCEPT are evaluated in order of appearance
- Parenthesis always have the highest order of precedence so they can be used to control the order of evaluation

```
-- Returns locations that are supplier locations, but not locations that are both employee
and customer locations
SELECT
    country,
    region,
    city
FROM Production.Suppliers

EXCEPT -- Runs second, with the second query now being the result set of the intersect
operator

SELECT
    country,
    region,
    city
FROM HR.Employees

INTERSECT -- Runs first

SELECT
    country,
    region,
    city
FROM Sales.Customers

-- Forcing precedence with parentheses, returns (locations that are supplier locations but
not employee locations) and that are also customer locations
(SELECT
    country,
    region,
    city
FROM Production.Suppliers

EXCEPT -- Runs first

SELECT
    country,
    region,
```

```
    city
FROM HR.Employees)

INTERSECT -- Runs second with the first query being the result set of the except operator

SELECT
    country,
    region,
    city
FROM Sales.Customers
```

Ch 7: Beyond the Fundamentals of Querying

Window Functions

- Function that computes a scalar value for each row, based on a calculation against a subset of the rows from the underlying query
    - The subset of rows is known as a window
- Typically, aggregations are applied to grouped queries (GROUP BY) or subqueries which presents with disadvantages
    - GROUP BY clauses lose detail, causing all computations in the query to be done in context of the defined group
- *A window function is evaluated per detailed row, and it's applied to a subset of rows that is derived from the underlying query result set. The result of the window function is a scalar value, which is added as another column to the query result.*
- Another benefit to window functions is the ability to define order as part of the specification of the calculation (when applicable)
- There are three parts of a window function, all specified in a clause called OVER ()
    - Window-Partition Clause PARTITION BY ()
        - Restricts the window to the subset of rows that have the same values in the partitioning columns as in the current row
    - Window-Order Clause ORDER BY ()
        - Defines ordering within the window function, not to be confused with presentation order
    - Window-Frame Clause ROWS BETWEEN <top delimiter> AND <bottom delimiter>
        - Filters between the two specified delimiters

        - *Use UNBOUNDED PRECEEDING as the top delimiter to indicate that the window starts at the first row of the parition*
        - *Use UNBOUNDED FOLLOWING as the bottom delimiter to indicate that the window ends at the last row of the partition*
- Window functions are only allowed in the SELECT and ORDER BY clauses of a query
    - There are exceptions but these are beyond the scope of this book
- Functions that be used with a window function
    - Aggregate functions
        - COUNT
        - SUM
        - AVG
        - MIN
        - MAX
    - Offset functions
        - FIRST_VALUE
        - LAST_VALUE
        - LEAD
        - LAG
    - Statistical functions
        - PERCENT_RANK
        - CUME_DIST
        - PERCENTILE_CONT
        - PERCENTILE_DISC

```
-- Basic Window Function
SELECT
    empid,
    ordermonth,
    val,
    SUM (val) OVER (
        PARTITION BY empid
        ORDER BY ordermonth
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW -- No Boundaries
    ) as runval
FROM Sales.EmpOrders
```

Ranking Window Functions

- Used to rank each row with respect to others in the window
- T-SQL supports the following ranking functions;
  - ROW_NUMBER (): Assigns incremental sequential integers to the rows in the query result based on the mandatory window ordering
    - If the ORDER BY list is not unique, the query is nondertiministic (more than one correct answer is possible)
    - You would need to add a tiebreaker to the ORDER BY list to make it unique
  - RANK (): Reflects the count of rows that have a lower ordering value than the current row (plus 1), non-distinct values receive the same rank
  - DENSE_RANK (): Reflects the count of distinct ordering values that are lower than the current row (plus 1), does not skip ranks
  - NTILE (): Associate the rows in the result with the specified tiles (equally sized groups of rows)
- Ranking functions can be used with a window partition clause, restricting the window to only those rows that have the same values in the partitioning attributes as in the current row
- Window functions are logically evaluated in the SELECT clause before DISTINCT (if applicable)

```
-- All Ranking Functions
SELECT orderid, custid, val,
    ROW_NUMBER()    OVER (ORDER BY val) AS rownum,
    RANK ()         OVER (ORDER BY val) AS rank,
    DENSE_RANK ()   OVER (ORDER BY val) AS dense_rank,
    NTILE (10)     OVER (ORDER BY val) AS ntile
FROM Sales.OrderValues
ORDER BY val;

-- RANK () with PARTITION BY
SELECT orderid, custid, val,
    ROW_NUMBER()    OVER (PARTITION BY custid ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

Offset Window Functions

- Returns an element from a row that is at a certain offset from the current row or window frame
- LAG () & LEAD ()

    - Supports window partitions and window order clauses
    - Used to obtain an element from a row that is a certain offset from the current row within a partition (based on indicated ordering)
    - Arguments: Element to Return, Offset (Default = 1), Value if NULL (Default = NULL)
- FIRST_VALUE () & LAST_VALUE()
    - Supports window partitions, window order, and window frame clauses (window frame should be explicit, ROWS BETWEEN...)
    - Used to return an element from the first and last rows in the window frame

```
-- LAG () & LEAD ()
SELECT custid, orderid, val,
    LAG (val)   OVER (PARTITION BY custid ORDER BY orderdate, orderid) AS preval, --
Returns the previous order value, second and third argument DEFAULT
    LEAD (val)  OVER (PARTITION BY custid ORDER BY orderdate, orderid) AS nextval --
Returns the next order value, second and third argument DEFAULT
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;

-- Computes the difference between a customers previous order and the current row order
SELECT custid, orderid, val,
    val - LAG (val) OVER (PARTITION BY custid ORDER BY orderdate, orderid) AS preval
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;

-- FIRST_VALUE () & LAST_VALUE ()
SELECT custid, orderid, val,
    FIRST_VALUE (val)   OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS firstval, -- Returns the first value in the window
    LAST_VALUE (val)    OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS BETWEEN
CURRENT ROW AND UNBOUNDED FOLLOWING) AS lastval -- Returns the last value in the window
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;

-- Computes the difference between the current row value and the first and last value for
each customer
SELECT custid, orderid, val,
    FIRST_VALUE (val)       OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS firstval,
    LAST_VALUE (val)        OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lastval,
    val - FIRST_VALUE (val) OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS firstval, -- differnece
    val - LAST_VALUE (val)  OVER (PARTITION BY custid ORDER BY orderdate, orderid ROWS
BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lastval -- difference
```

```
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;
```

Aggregate Window Functions

- Used to aggregate the rows in a defined window
- Using OVER () with no parameters exposes a window of all rows from the underlying query's result set to the specified aggregate function
- Using PARTITION BY exposes a restricted window to the specified aggregate function
- Using window frames with aggregate functions allows for more sophisticated calculations
    - Running and moving aggregates, YTD and MTD calculations, etc.

```
-- Aggregate on open and restricted window
SELECT custid, orderid, val,
    SUM(val)    OVER () AS totalvalue, -- Returns the total sum of ALL orders
    SUM(val)    OVER (PARTITION BY custid) AS custtotalvalue -- Returns the total sum of
the customers orders
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;

-- Returning percentages of total values
SELECT custid, orderid, val,
    100.0 * val / SUM (val) OVER () AS totalvalue, -- Returns the percentage of the rows
order value relative to the sum of ALL orders
    100.0 * val / SUM (val) OVER (PARTITION BY custid) AS custtotalvalue --  Returns the
percentage of the rows order value relative to the sum of the customers orders
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;

-- Using a window frame with an aggregate function
SELECT empid, ordermonth, val,
    SUM (val)   OVER (PARTITION BY empid ORDER BY ordermonth ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS runval -- returns a running total of all orders for each
employee id
FROM Sales.EmpOrders;
```

Pivoting Data

- Involves rotating data from a state of rows to a state of columns, possibly aggregating values along the way

    - This is usually handled by the presentation layer for reporting (excel or similar)
- Every pivoting request involves three logical processing phases;
    - Grouping phase with associated grouping or on rows element
    - Spreading phase with associated spreading or on cols element
    - Aggregation phase with association aggregation element and function
- The below example will group by employee id, spread quantities by customer id, and aggregate quantity with the SUM function
    - Two solutions are presented;
        - Pivoting with a grouped query
        - Pivoting with the PIVOT operator

```
-- Creating temporary table for demonstration
USE TSQLV4;

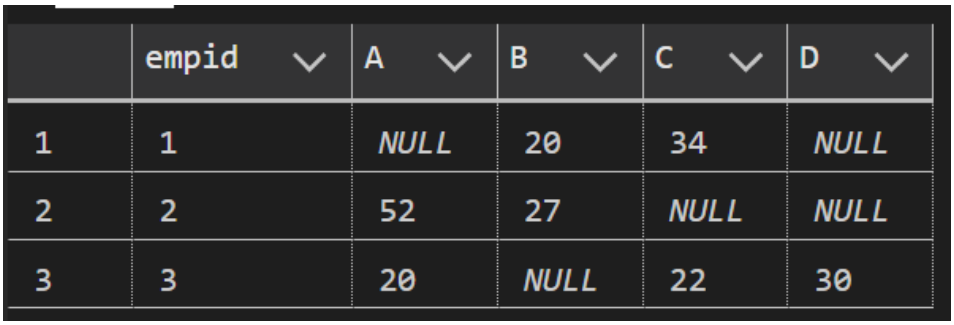
DROP TABLE IF EXISTS dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid     INTEGER     NOT NULL,
    orderdate   DATE        NOT NULL,
    empid       INTEGER     NOT NULL,
    custid      VARCHAR(5)  NOT NULL,
    qty         INTEGER     NOT NULL
    CONSTRAINT  PK_Orders   PRIMARY KEY (orderid)
);

INSERT INTO dbo.Orders (orderid, orderdate, empid, custid, qty)
VALUES
    (30001, '20140802', 3, 'A', 10),
    (10001, '20141224', 2, 'A', 12),
    (10005, '20141224', 1, 'B', 20),
    (40001, '20150109', 2, 'A', 40),
    (10006, '20150118', 1, 'C', 14),
    (20001, '20150212', 2, 'B', 12),
    (40005, '20160212', 3, 'A', 10),
    (20002, '20160216', 1, 'C', 20),
    (30003, '20160418', 2, 'B', 15),
    (30004, '20140418', 3, 'C', 22),
    (30007, '20160907', 3, 'D', 30);

SELECT * FROM dbo.Orders;
```

Pivoting with a Grouped Query

- The grouping phase is achieved with a GROUP BY clause on employee id
- The spreading phase is achieved in the SELECT clause with a CASE expression for each target, customer id in this case
- The aggregation phase is achieved by applying the relevant aggregate function to the result of the CASE expression
- This returns the quantity from the current row only when the current row represents an order for the relevant customer id, otherwise the expression returns null
    - If an ELSE clause is not specified in a CASE expression, the implied default is ELSE NULL
    - 

	empid	A	B	C	D
1	1	*NULL*	20	34	*NULL*
2	2	52	27	*NULL*	*NULL*
3	3	20	*NULL*	22	30

```
-- Pivot with a grouped query
SELECT
    empid,
    SUM (CASE WHEN custid = 'A' THEN qty END) AS A,
    SUM (CASE WHEN custid = 'B' THEN qty END) AS B,
    SUM (CASE WHEN custid = 'C' THEN qty END) AS C,
    SUM (CASE WHEN custid = 'D' THEN qty END) AS D
FROM dbo.Orders
GROUP BY empid;
```

Pivoting with the PIVOT Operator

- PIVOT operates in the FROM clause similar to other table operators
- It operates on a source table or table expression (subquery) provided to it as its left input, pivots the data, and returns a result table
- The PIVOT operator involves the same logical processing phases as above (grouping, spreading, aggregating)
- The PIVOT operator does not require a grouping element to be explicitly specified, it figure out which element to group by elimination
    - Because of this, it is recommended to use a subquery instead of the entire source table, specifying the exact columns needed for the PIVOT phase;

- The grouping element (employee id)
- The spreading element (customer id)
- The aggregation element (SUM on quantity)

```
-- General Syntax
SELECT ...
FROM input_table
    PIVOT (
        aggregate_function (aggregate_element)
        FOR spreading_element IN (list_of_target_columns)
    ) AS result_table_alias;

-- Pivot with PIVOT operator (same result as above)
SELECT empid, A, B, C, D
FROM (
    SELECT empid, custid, qty
    FROM dbo.Orders
) AS D
    PIVOT (
        SUM (qty)
        FOR custid IN (A, B, C, D)
    ) AS P;

-- Alternate example grouping by customer id and spreading by employee id
SELECT custid, [1], [2], [3]
FROM (
    SELECT empid, custid, qty
    FROM dbo.Orders
) AS D
    PIVOT (
        SUM (qty)
        FOR empid IN ([1], [2], [3]) -- employee id identifier are irregular so must be
delimited in brackets
    ) AS P;
```

Unpivoting Data

- Involves rotating data from a state of columns to a state of rows
  - Commonly used to unpivot data imported from a spreadsheet
- Every unpivoting request involves three logical processing phases;
  - Producing copies
  - Extracting values
  - Eliminating irrelevant rows
- Two solutions are presented;

    - Using the APPLY operator
    - Using the UNPIVOT operator

```
-- Creating temporary table for demonstration
USE TSQLV4;

DROP TABLE IF EXISTS dbo.EmpCustOrders;

CREATE TABLE dbo.EmpCustOrders
(
    empid   INT         NOT NULL    CONSTRAINT PK_EmpCustOrders PRIMARY KEY,
    A       VARCHAR(5)  NULL,
    B       VARCHAR(5)  NULL,
    C       VARCHAR(5)  NULL,
    D       VARCHAR(5)  NULL,
);

INSERT INTO dbo.EmpCustOrders (empid, A, B, C, D)
    SELECT empid, A, B, C, D
    FROM (
        SELECT empid, custid, qty
        FROM dbo.Orders
    ) AS D
        PIVOT (
            SUM (qty)
            FOR custid IN (A, B, C, D)
        ) AS P;

SELECT * FROM dbo.EmpCustOrders;
```

Unpivoting with the APPLY Operator

- The first step involves producing multiple copies of each source row, one for each column you need to unpivot
    - This is done by using the CROSS APPLY function and generating a virtual temporary table with VALUES
    - CROSS APPLY is used instead of CROSS JOIN because the former evaluates the left side first and then applies the right side to each left row
- The second step involves extracting a value from the original customer columns (A, B, C, D) to return a single value column (quantity)
- The third step involves eliminating NULL values (this is optional)
    - In the original table, NULLs represnt irrelevant interesections and there is no reason to keep these

- This is accomplised by adding a WHERE clause that filters IS NOT NULL

```
-- Unpivoting with APPLY operator
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
    CROSS APPLY (
        VALUES ('A', A), ('B', B), ('C', C), ('D', D)
    ) AS C (custid, qty)
WHERE qty IS NOT NULL; -- WHERE clause is optional
```

Unpivoting with the UNPIVOT operator

- Involves producing two result columns from any number of source columns
- Implemented as a table operator or table expression (subquery)
- The UNPIVOT operator involves the same logical processing phases as above (generating copies, extracting elements, eliminating irrelevant rows)
- The third phase is required for the UNPIVOT operator, unlike the APPLY operator
  - If you need to keep NULL values, use the APPLY operator
  - Oterwise, use the UNPIVOT operator

```
-- General Syntax
SELECT ...
FROM input_table
    UNPIVOT (
        values_column
        FOR names_column IN (list_of_names_columns)
    ) AS result_table_alias
WHERE ...;

-- Unpivot with UNPIVOT operator (same result as above when NULLs are excluded)
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
    UNPIVOT (
        qty
        FOR custid IN (A, B, C, D)
    ) AS U;
```

Grouping Sets

- Set of expressions grouped in a grouped query (GROUP BY clause)
- Single grouping sets can be *combined* using ;
  - GROUPING SETS, CUBE, and ROLLUP sub-clauses

    - GROUPING function
- Alternatively, all the below queries can be combined with UNION but that is cumbersome and prone to errors and performance issues

```
-- Examples of Single Grouping Sets
SELECT empid, custid, SUM (qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid; -- grouping set empid and custid

SELECT empid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid; -- grouping set empid

SELECT custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY custid; -- grouping set custid

SELECT SUM(qty) AS sumqty
FROM dbo.Orders; -- grouping set empty ()
```

The GROUPING SETS Sub-Clause

- Used to define multiple grouping sets in the same query
- The attributes are listed in the GROUPING SETS sub-clause
- This is a logical equivalent of combining the above queries with UNION, but cleaner and more performant

```
-- GROUPING SETS Sub-Clause
SELECT
    empid,
    custid,
    SUM (qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS (
        (empid, custid), -- first grouping set
        (empid), -- second grouping set
        (custid), -- third grouping set
        () -- fourth grouping set (empty)
    );
```

The CUBE Sub-Clause

- Provides an abbreviated way to define multiple grouping sets

- The result is all possible combinations of the grouping sets passed into the CUBE sub-clause
    - Example: CUBE (A, B, C) produces eight grouping sets: (A, B, C), (A, B), (A, C), (B, C), (A), (B), (C), ()
- This is a logical equivalent of both above queries

```
-- CUBE Sub-Clause
SELECT
    empid,
    custid,
    SUM (qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE (empid, custid);
```

The ROLLUP Sub-Clause

- Similarly provides an abbreviated way to define multiple grouping sets, however ROLLUP does not provide all possible combinations of grouping sets
- ROLLUP assumes a hierarchy among the input attributes and only produces grouping sets with leading combinations
    - Example: ROLLUP (A, B, C) produces four grouping sets: (A, B, C), (A, B), (A), ()
- A possible use case is returning some aggregation for all grouping sets based on a date hierarchy (year, month, day)

```
-- ROLLUP Sub-Clause
SELECT
    YEAR (orderdate) AS orderyear,
    MONTH (orderdate) AS ordermonth,
    DAY (orderdate) AS orderday,
    SUM (qty) AS sumqty
FROM dbo.Orders
GROUP BY ROLLUP (YEAR (orderdate), MONTH (orderdate), DAY (orderdate));
```

The GROUPING Function

- Function accepts a name of a column and returns 0 if it is a member of the current grouping set, and 1 if it is not a member of the current grouping set
- This allows you to see if NULLs are placeholders or if the value is an actual NULL from the source data

```
-- GROUPING Function
```

```
SELECT
    GROUPING (empid) AS groupemp,
    GROUPING (custid) AS groupcus,
    empid,
    custid,
    SUM (qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE (empid, custid);
```

Ch 8: Data Modification

- SQL has a set of statements known as Data Manipulation Language (DML) that deal with data manipulation
- DML includes statements SELECT, INSERT, UPDATE, DELETE, TRUNCATE, and MERGE

Inserting Data

- T-SQL provides several statements for inserting data into tables
  - INSERT VALUES, INSERT SELECT, INSERT EXEC, SELECT INTO, and BULK INSERT

The INSERT VALUES Statement

- Used to insert rows into a table based on specified values
- Specifying the target column names right after the table name is optional, but advised
  - Doing so allows you to control the value-column association instead of relying on the column order specified in the CREATE TABLE statement
- If a value for a column is not specified, a default value will be used if one is defined (such as orderdate)
  - If there is not default value and the column allows NULLs, a NULL will be inserted
  - Otherwise the INSERT statement will fail
- When inserting multiple rows, the query is processed as a transaction
  - If any row fails, the entire statement will fail and no rows will be inserted
- VALUES can be used by itself to craete a derived table (temporary table)
  - Following the parenthesis you assign a table allias and column aliases

```
-- Create 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
);

-- Insert Single Row
INSERT INTO dbo.Orders (
    orderid,
    orderdate,
    empid,
    custid
)
    VALUES (10001, '2016-02-12', 3, 'A');

-- Insert Single Row w/ Default Values
INSERT INTO dbo.Orders (
    orderid,
    -- orderdate will insert as default SYSTIMEDATE()
    empid,
    custid
)
    VALUES (10002, 5, 'B');

-- Insert Multiple Rows
INSERT INTO dbo.Orders (
    orderid,
    orderdate,
    empid,
    custid
)
    VALUES
        (10003, '2016-02-13', 4, 'B'),
        (10004, '2016-02-14', 1, 'A'),
        (10005, '2016-02-13', 1, 'C'),
        (10006, '2016-02-15', 3, 'C');

-- Inserting Values Into Derived Table (Temp Table)
SELECT *
FROM (
```

```
    VALUES
        (10003, '2016-02-13', 4, 'B'),
        (10004, '2016-02-14', 1, 'A'),
        (10005, '2016-02-13', 1, 'C'),
        (10006, '2016-02-15', 3, 'C')
) AS O (orderid, orderdate, empid, custid); -- table and column aliases
```

The INSERT SELECT Statement

- Inserts a set of rows returned by a SELECT query into a target table
- Syntax is similar to an INSERT VALUES statement but a SELECT query is specified in place of a VALUES clause
- The behavior in terms of default values and column nullability is the same as the INSERT VALUES statement
- The INSERT SELECT statement is performed as a transaction, so If any row fails, the entire statement will fail and no rows will be inserted

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

The INSERT EXECUTE Statement (EXEC)

- Inserts a set of rows returned by an executed stored procedure or a dynamic SQL batch into a target table
    - Stored procedures are reviewed in-depth in Chapter 11: Programmable Objects
- Syntax is similar to an INSERT VALUES statement but a EXECUTE statement is specified in place of a VALUES clause

- The stored procedure and parameter are passed into the EXECUTE statement and the result set is inserted into the table

```
-- Create Stored Procedure (Reviewed in Ch 11)
DROP PROCEDURE IF EXISTS Sales.GetOrders;
GO
CREATE PROCEDURE Sales.GetOrders
    @country AS NVARCHAR(40)
AS
SELECT
    orderid,
    orderdate,
    empid,
    custid
FROM Sales.Orders
WHERE shipcountry = @country;
GO

-- Test Stored Procedure
EXECUTE Sales.GetOrders @country = N'France'; -- results in all orders from France

-- Inserting a Result Set from a Stored Procedure with INSERT EXEC
INSERT INTO dbo.Orders (
    orderid,
    orderdate,
    empid,
    custid
)
    EXECUTE Sales.GetOrders @country = N'France'; -- inserts results from stored procedure
into target table
```

The SELECT INTO Statement

- Non-standard T-SQL statement that creates a target table and populates it with the result set of a query
- *You cannot use this statement to insert data into an existing table*
- Syntax: Add INTO <target_table_name> right before the FROM clause of a query
- The target table's structure and data are based on the source table
    - Source column names, data types, nullability, and identity properties are copied
    - Source constraints, indexes, triggers, column properties, and permissions are *NOT* copied

- SELECT INTO can also be used with set operations such as the EXCEPT operation (last example query)

```
-- Drop and Create New Table with SELECT INTO
DROP TABLE IF EXISTS dbo.Orders;

SELECT
    orderid,
    orderdate,
    empid,
    custid
INTO dbo.Orders -- new table
FROM Sales.Orders; -- source table

-- Using SELECT INTO with EXCEPT
DROP TABLE IF EXISTS dbo.Locations;

SELECT
    country,
    region,
    city
INTO dbo.Locations
FROM Sales.Customers

EXCEPT

SELECT
    country,
    region,
    city
FROM HR.Employees; -- inserts locations that are customer locations but not employee
locations
```

The BULK INSERT Statement

- Used to insert data from a file into an exisiting table
- The statement specified the target table, source file, and options
    - Options include the data file type, the field terminator (seperates columns), the row terminator, etc.

```
-- Inserting Rows from File Using BULK INSERT
BULK INSERT dbo.Orders FROM 'file_location'
    WITH (
        DATAFILETYPE    = 'char',
        FIELDTERMINATOR = ',',
```

```
        ROWTERMINATOR   ='\n'
    );
```

The Identity Property and Sequence Object

- SQL Server supports two built-in solutions to automatical generate numeric keys, the identiy column object and the sequence object

Identity Property

- Identity is a standard column property used for a column with any numeric type with a scale of zero (no fraction)
- When defining the property, you can optionally specify a seed (first value) and an increment (step value)
    - If not provided, the default is one for both
    - This would result in 1, 2, 3, 4...
- This property is generally used to generate a surrogate key produced by the system, not derived from application data
- When a new row is inserted, SQL Server generates a new identity value based on the current value in the table and the increment
- When inserting row(s), the identity column must be ignored unless IDENTITY_INSERT is used
    - Duplicate values can be manually inserted for the identity column unless it is set up with a primary key or unique constraint
- The change to the current identity value in a table is not undone if the INSERT statement fails
    - Meaning if the current identity value is 5, and you attempt to insert a single row but it fails, the new identity value will still be 6
    - The identity property should only be used if you can allow gaps between keys, otherwise you have to implement a different mechanism to generate keys
- One of the shortcomings of the identity column is you cannot add it to an existing column or remove it from an existing column

```
-- Create Table with Identity Column Property
DROP TABLE IF EXISTS dbo.T1;

CREATE TABLE dbo.T1 (
```

```
    keycol  INT         NOT NULL    IDENTITY (1, 1) CONSTRAINT PK_T1            PRIMARY
KEY, -- the start value is 1, and the key will increment by 1
    datacol VARCHAR(10) NOT NULL                    CONSTRAINT CHK_T1_datacol
CHECK(datacol LIKE '[A-Z]%') -- only allows values inserted that start with a letter A
through Z
);

-- Inserting Rows
INSERT INTO dbo.T1 (
    datacol
)
    VALUES
        ('A'),
        ('B'),
        ('C'); -- SQL Server automatically generates the identity column values (keycol in
this case)

-- Inserting Rows with Explicit Values for Identity Column Using IDENTITY INSERT
SET IDENTITY_INSERT dbo.T1 ON;

INSERT INTO dbo.T1 (
    keycol,
    datacol
)
    VALUES
        (4, 'D'),
        (5, 'E');
```

Sequence Object

- Alternative key generating mechanism for identity, more flexible and preferred
- One of the advantages over the identity property is that it is not tied to a particular column in a particular table, 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
- To create a sequence object, you use the CREATE SEQUENCE command, below are optional properties;
    - Data type can be any numeric type with a scale of zero (no fraction) and is indicated with AS <data-type>
        - If not specified, the default is BIGINT
    - Minimum and maximum values can be indicated with MINVALUE <value> and MAXVALUE <value>

        - If not specified, the defaults is the minimum and maximum values of whatever data type is specified
    - Cycling is supported and must be indicated with CYCLE
        - If not specified, the default is NO CYCLE
    - Starting values and increment can be indicated with START WITH <value> and INCREMENT BY <value>
        - If not specified, the default start value will be the minimum value of the data type, and the default increment will be 1
    - Caching option can be indicated with CACHE <value> or NO CACHE
        - If not specified, the default is CACHE 50
- You can change any of the sequence properties except the data type with ALTER SEQUENCE
- To generate a new sequence value, you invoke the standard function NEXT VALUE FOR <sequence_name>
    - You don't need to insert a row into a table in order to generate new values
- With sequences, you can store the result of the function in a variable and use it later in your code (such as INSERT INTO to insert rows)
    - Alternatively, you can use the NEXT VALUE FOR clause directly in the VALUES statement
- You can generate new sequence values in an UPDATE statement to replace the existing values
- Similar to the identity property, the sequence object does not guarantee you will have no gaps
    - If a new sequence value was generated by a transaction that failed or is intentianally rolled back, the sequence change is not undone

```
-- Create Sequence
CREATE SEQUENCE dbo.SeqOrderIDs AS INT
    MINVALUE        1
    START WITH      1
    INCREMENT BY    1
    CYCLE; -- any parameters not specified will be default

-- Alter Sequence
ALTER SEQUENCE dbo.SeqOrderIDs
    NO CYCLE;

-- Generating New Sequence Value
SELECT NEXT VALUE FOR dbo.SeqOrderIDs; -- returns 1 (do not actually run)
```

```
-- Creating a Table and Inserting Values with a Declared Variable and Sequence
DROP TABLE IF EXISTS dbo.T1;

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

DECLARE @neworderid AS INT = NEXT VALUE FOR dbo.SeqOrderIDs;

INSERT INTO dbo.T1 (
    keycol,
    datacol
)
    VALUES (@neworderid, 'A');

-- Using NEXT VALUE FOR in the VALUES Statement
INSERT INTO dbo.T1 (
    keycol,
    datacol
)
    VALUES
        (NEXT VALUE FOR dbo.SeqOrderIDs, 'B'),
        (NEXT VALUE FOR dbo.SeqOrderIDs, 'C'),
        (NEXT VALUE FOR dbo.SeqOrderIDs, 'D');

-- Updating Existing Sequence Values
UPDATE dbo.T1
    SET keycol = NEXT VALUE FOR dbo.SeqOrderIDs;

-- Querying Information About Sequence
SELECT current_value
FROM sys.sequences
WHERE OBJECT_ID = OBJECT_ID(N'dbo.SeqOrderIDs');

-- Clean Up
DROP TABLE IF EXISTS dbo.T1;
DROP SEQUENCE IF EXISTS dbo.SeqOrderIDs;
```

Deleting Data

- T-SQL provides two statements for deleting rows from a table: DELETE and TRUNCATE
- Both the TRUNCATE and DELETE statements are transactional
- Run the following code to create copies of the Customers and Orders table

```
-- Create Temporary Tables for Demonstration
DROP TABLE IF EXISTS dbo.Orders, dbo.Customers;

CREATE TABLE dbo.Customers (
  custid       INT          NOT NULL CONSTRAINT PK_Customers PRIMARY KEY(custid),
  companyname  NVARCHAR(40) NOT NULL,
  contactname  NVARCHAR(30) NOT NULL,
  contacttitle NVARCHAR(30) NOT NULL,
  address      NVARCHAR(60) NOT NULL,
  city         NVARCHAR(15) NOT NULL,
  region       NVARCHAR(15) NULL,
  postalcode   NVARCHAR(10) NULL,
  country      NVARCHAR(15) NOT NULL,
  phone        NVARCHAR(24) NOT NULL,
  fax          NVARCHAR(24) NULL,
);

CREATE TABLE dbo.Orders (
  orderid        INT          NOT NULL  CONSTRAINT PK_Orders            PRIMARY
KEY(orderid),
  custid         INT          NULL      CONSTRAINT FK_Orders_Customers  FOREIGN
KEY(custid) REFERENCES dbo.Customers(custid),
  empid          INT          NOT NULL,
  orderdate      DATE         NOT NULL,
  requireddate   DATE         NOT NULL,
  shippeddate    DATE         NULL,
  shipperid      INT          NOT NULL,
  freight        MONEY        NOT NULL  CONSTRAINT DFT_Orders_freight   DEFAULT(0),
  shipname       NVARCHAR(40) NOT NULL,
  shipaddress    NVARCHAR(60) NOT NULL,
  shipcity       NVARCHAR(15) NOT NULL,
  shipregion     NVARCHAR(15) NULL,
  shippostalcode NVARCHAR(10) NULL,
  shipcountry    NVARCHAR(15) NOT NULL,
);
GO

-- Copy Data From Sales.Customers and Sales.Orders
INSERT INTO dbo.Customers SELECT * FROM Sales.Customers;
INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

The DELETE Statement

- Used to delete from a table based on an optional filter predicate
- The FROM clause specified the table

- The WHERE clause specifies the predicate
- Only the subset of rows for which the predicate evaluates to TRUE will be deleted
- The DELETE statement is a fully logged operation so it tends to be expensive when a large number of rows are deleted

```
-- Deleting Filtered Records
DELETE FROM dbo.Orders
WHERE orderdate < '2015-01-01';
```

The TRUNCATE Statement

- Used to delete ALL rows from a table (no filter predicate)
- The TRUNCATE statement is minimally logged, resulting in significant performance differences when compared to the DELETE statement
    - Despite being minimally logged, SQL Server records which blocks of data were deallocated by the operations so that they can be reclaimed in case the operation needs to be undone
- The TRUNCATE statement is not allowed when the target table is referenced by a foreign key, even if the table is empty and/or the foreign key is disabled
    - All foreign keys would need to be dropped before truncating with the ALTER TABLE DROP CONSTRAINT command
    - Foreign keys can be re-created after truncating with the ALTER TABLE ADD CONSTRAINT command

```
-- Deleting All Records
ALTER TABLE dbo.Orders
DROP CONSTRAINT FK_Orders_Customers; -- removing foreign key constraints from orders on
customers

TRUNCATE TABLE dbo.Customers;
```

DELETE Based on a Join

- T-SQL supports a nonstandard DELETE 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

    - The first clause that is logically processed is the FROM clause, followed by the WHERE clause, and finally the DELETE clause
    - The DELETE clause references the table alias of the side of the join that is supposed to be the target table for the deletion
- A subquery can be used to accomplish the same task, and is considered standard
- SQL Server processes both the same way so there is no performance difference, but it is a best practice to stick with the standard

```
-- Delete w/ Join (Non-Standard)
DELETE FROM O
FROM dbo.Orders AS O
INNER JOIN dbo.Customers AS C
    ON O.custid = C.custid
WHERE C.Country = N'USA';

-- Delete w/ Subquery (Standard)
DELETE FROM dbo.Orders
WHERE EXISTS (
    SELECT *
    FROM dbo.Customers AS C
    WHERE Orders.custid = C.custid
        AND C.country = N'USA'
);

-- Clean Up
DROP TABLE IF EXISTS dbo.Orders, dbo.Customers;
```

Updating Data

- T-SQL supports a standard UPDATE statement and non-standard forms of the update statement with joins and with variables
- Run the following code to create copies of the Customers and Orders table

```
-- Create Temporary Tables for Demonstration
DROP TABLE IF EXISTS dbo.OrderDetails, dbo.Orders;

CREATE TABLE dbo.Orders (
    orderid        INT          NOT NULL    CONSTRAINT PK_Orders PRIMARY KEY(orderid),
    custid         INT          NULL,
    empid          INT          NOT NULL,
    orderdate      DATE         NOT NULL,
    requireddate   DATE         NOT NULL,
    shippeddate    DATE         NULL,
    shipperid      INT          NOT NULL,
```

```
    freight        MONEY        NOT NULL    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname       NVARCHAR(40) NOT NULL,
    shipaddress    NVARCHAR(60) NOT NULL,
    shipcity       NVARCHAR(15) NOT NULL,
    shipregion     NVARCHAR(15) NULL,
    shippostalcode NVARCHAR(10) NULL,
    shipcountry    NVARCHAR(15) NOT NULL,
);

CREATE TABLE dbo.OrderDetails (
    orderid       INT           NOT NULL    CONSTRAINT FK_OrderDetails_Orders FOREIGN
KEY(orderid) REFERENCES dbo.Orders(orderid),
    productid     INT           NOT NULL    CONSTRAINT PK_OrderDetails PRIMARY
KEY(orderid, productid),
    unitprice     MONEY         NOT NULL    CONSTRAINT DFT_OrderDetails_unitprice
DEFAULT(0),           CONSTRAINT CHK_unitprice CHECK (unitprice >= 0),
    qty           SMALLINT      NOT NULL    CONSTRAINT DFT_OrderDetails_qty DEFAULT(1),
              CONSTRAINT CHK_qty  CHECK (qty > 0),
    discount      NUMERIC(4, 3) NOT NULL    CONSTRAINT DFT_OrderDetails_discount
DEFAULT(0),            CONSTRAINT CHK_discount  CHECK (discount BETWEEN 0 AND 1),
);
GO

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;
```

The UPDATE Statement

- Standard statement used to update a subset of rows in a table
- Specify a predicate in the WHERE clause to identify the subset of rows you need to update
- The assignment of updated values is specified in a SET clause, seperated by commas
    - T-SQL supports compound assignment operators;
        - += plus equal
        - -= minus equal
        - *= multiplication equal
        - /= division equal
        - %= modulo equal

```
-- Updating Records
UPDATE dbo.OrderDetails
    SET discount += 0.05
```

```
WHERE productid = 51;
```

UPDATE Based on a Join

- T-SQL supports a nonstandard UPDATE 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 update rows from one table based on a filter against attributes in related rows from another table
  - The first clause that is logically processed is the FROM clause, followed by the WHERE clause, and finally the UPDATE clause
  - The UPDATE clause references the table alias of the side of the join that is supposed to be the target table for the update
- A subquery can be used to accomplish the same task, and is considered standard
- SQL Server processes both the same way so there is no performance difference, but it is a best practice to stick with the standard

```
-- Update w/ Join (Non-Standard)
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 w/ Subquery (Standard)
UPDATE dbo.OrderDetails
    SET discount += 0.05
WHERE EXISTS (
    SELECT *
    FROM dbo.Orders AS O
    WHERE O.orderid = OrderDetails.orderid
        AND O.custid = 1
);

-- Clean Up
DROP TABLE IF EXISTS dbo.Orders, dbo.OrderDetails;
```