Module 2: SELECT, WHERE, and ORDER BY

Column Aliases

At the end of the previous module we presented the following query:

```
SELECT
	CarID
	,'Great Car!'
	,Make
	,Model
	,Year
FROM dbo.Cars;
```

In the result set notice that no column name appears for the column containing the 'Great Car!' values. We can fix that by adding a column alias.

In fact, we can provide a column alias for any column, as seen below.

```
SELECT
	CarID as [The Car ID]
	,'Great Car!' as [Great Car or Greatest Car]
	,Make as [Car Make]
	,Model as [Car Model]
	,Year as [Car Year]
FROM dbo.Cars;
```

There are different ways of creating a column alias. The query below uses single ticks to delineate the column alias, and the one below that doesn't use the AS keyword. We're going to use the AS keyword and brackets throughout this course to create column aliases.

```
SELECT
	CarID
	,'Great Car!' AS 'Great Car or Greatest Car?!'
	,Make AS 'Car Make'
	,Model
	,Year
FROM dbo.Cars;
```

As a general rule, you want to make sure that columns always have a name, and a column alias is how you can do that.

Column Expressions, Data Types, and Built-in Functions

In addition to column names and literal values, we can use expressions in the SELECT list as well. Generally speaking, as long as the expressions returns a scalar (i.e. discrete) value, we can use it.

For example, let's say we want to return a list of names from the dbo.Person table. In addition to the separate **FirstName** and **LastName** columns, perhaps we'd like to return a **FullName** column. We can do this using an ***expression***. The + sign is the addition operator when used with numbers, but the string concatenation operator when used with strings. Consider the following examples of expressions:

```
SELECT
	FirstName
	,LastName
	,LastName + ', ' + FirstName as [Full Name]
FROM [dbo].[Person];


SELECT
	SaleID
	,SalesDate
	,SalesAmount
	,SalesAmount + 100 as [Sales Amount plus 100]
FROM dbo.Sales;
```

The results in the first query include a third column named Full Name containing the LastName value, and a comma and a space, and the FirstName value. The second query simply adds the integer value 100 to each sales amount.

Data Types and Implicit Type Conversion

Built-in scalar functions can also be used in the SELECT list. TSQL contains many of these; a complete list can be found at this link (https://technet.microsoft.com/en-us/library/ms174318.aspx). Also, in TSQL all column values, variables, parameters, and literals have a specific **data type**. Expressions, including those that use functions, require that data types be compatible with the operations performed. Further, values used together within an expression must have the same data type.

The below query will return a data type conversion error, as we cannot add a Money value and a character string.

```
SELECT
	CarID
	,Make
	,Model
	,Year
	,DealerPrice
	,DealerPrice + 'Hello!' AS [DealerPrice plus 100]
FROM dbo.Cars;
```

The below query returns an error because we cannot divide SalesDate (a **Date** value) by SalesAmount (a Money value).

```
SELECT
	SaleID
	,SalesDate
	,SalesAmount
	,SalesDate / SalesAmount AS [Can we do this?]
FROM dbo.Sales;
```

Sometimes, if different data types are used together in an expression, the query processor will automatically convert the data type of one value to the data type of the other in order to perform the operation successfully. This is called an ***implicit type conversion***. Each SQL Server data type is assigned a rank ordered position in a list of **data type precedence**. In an expression, the value having the data

type with lower precedence is converted to the data type with higher precedence. The complete list of data type precedence can be found here (https://msdn.microsoft.com/en-us/library/ms190309.aspx).

Consider the following example:

```
SELECT
       PersonID
       ,FirstName
       ,LastName
       ,'Person ' + LastName AS [Implicit conversion to nvarchar]
FROM dbo.Person;
```

Lastname uses the **nvarchar(50)** data type. Nvarchar means a variable length character string, and the (50) means it can be up to 50 characters long. Nvarchar is also a ***Unicode string***. Unicode uses between two and four bytes for each character, and allows for the representation of most all of the world's languages.

The string literal 'Person' is a ***non-Unicode*** or ***standard string***. In this case, a single byte is used to store each character code. As a single byte can represent only 256 different characters, the **varchar()** data type can store a limited range of characters. In SQL Server the **char()** and **varchar()** data types are non-Unicode.

In the above query, the non-Unicode string is converted to a Unicode string in order to perform the string concatenation operation, and the result will therefore be a Unicode string.

This query puts an N in front of the string literal, making it a Unicode string literal:

```
SELECT
       PersonID
       ,FirstName
       ,LastName
       ,N'Person ' + LastName AS [Unicode string and result]
FROM dbo.Person;
```

The dbo.Person table also contains an **nchar(1)** data type for the MiddleInitial column. Nchar() indicates a fixed length string, and the (1) indicates that the string will be a single character in length. Notice that MiddleInitial also allows **NULL** values. NULL is a special value that means "unknown." We'll revisit NULL values at various times in the course.

Explicit Type Conversion with CAST and CONVERT

In the query below, the 3rd column in the result set is a money data type because the integer 2 is implicitly converted to a **MONEY** data type according to the data type precedence. The 4th and 5th columns demonstrate the use of **CAST** and **CONVERT**. In both cases, SalesAmount is explicitly converted to data type int, so when we multiply it by the integer value 2 the result is INT.

```
SELECT
       SaleID
       ,SalesDate
       ,SalesAmount * 2 AS [Result will be Money]
       ,CAST(SalesAmount AS INT) * 2 AS  [Result will be INT]
       ,CONVERT(INT,SalesAmount) * 2 AS [Result will also be INT]
FROM dbo.Sales;
```

There are some combinations of data types in which implicit type conversions will not occur. In these cases you need to explicitly convert one of the expressions with either the **CAST** or **CONVERT** function. While CAST and CONVERT have different syntax, in most situations these functions are equivalent.

Detailed information about CAST and CONVERT can be found here (https://technet.microsoft.com/en-us/library/ms187928.aspx), and a data type conversion chart showing all allowable implicit and explicit type conversions can be found here (https://www.microsoft.com/en-us/download/details.aspx?id=35834).

Date and Time Data Types and Functions

The date and time functions can be used with the values of the date and time data types. A complete listing of the date and time data types and functions can be found here (https://msdn.microsoft.com/en-us/library/ms186724.aspx).

The below query demonstrates the use of the DATE, TIME, DATETIME, and DATETIME2 data types, and the YEAR(), MONTH(), DAY(), GETDATE(), SYSDATETIME(), and DATEDIFF() functions.

```
SELECT
       SaleID
       ,SalesDate
       ,YEAR(SalesDate) AS [Year of sale]
       ,MONTH(SalesDate) AS [Month of sale]
       ,DAY(SalesDate) AS [Day of month of sale]
       ,GETDATE() AS [Current system date and time]
       ,SYSDATETIME() AS [Current system date and time]
       ,DATEDIFF(DAY,SalesDate,GETDATE()) AS [Days since sale was made]
       ,CAST(GETDATE() AS TIME) AS [Current system time without the date]
       ,CAST(GETDATE() AS DATE) AS [Current system time without the time]
FROM dbo.Sales;
```

IIF and CASE Logical Functions

Two logical functions you should know about include **IIF** and **CASE**. In fact, IIF can be considered a shorthand way of writing a CASE function. These are functions rather than control statements. Each takes input values and returns an output and does not control program flow.

IIF takes three input paramters:

1. A Boolean expression to test;
2. A value to return if the Boolean expression is true;
3. A value to return if the Boolean expression is not true.

The syntax is ***IIF (Boolean expression, value to return if true, value to return if not true)***. The query below uses IIF to determine whether the sales date was a Sunday. The datepart function returns the numeric day of the week of the sales date. Sunday is the first day of the week, so if datepart returns a 1, the day is a Sunday. If it returns anything else, it's not a Sunday.

```
SELECT
```

```
       s.SalesDate
       ,s.SalesAmount
       ,iif(datepart(dw,s.SalesDate) = 1,'Sunday','Not Sunday')
              as [Sold on Sunday?]
FROM dbo.Sales s
```

The below query uses a simple CASE function to determine whether the sales date was a weekday or a weekend day. CASE is followed by an input expression, which is the followed by one or more logical tests of the input expression. The first WHEN comparison that evaluates to true provide the returned value. If none match, ELSE provides the returned value.

```
SELECT
       s.SalesDate
       ,s.SalesAmount
       ,case datepart(dw,s.SalesDate)
              when 2 then 'Monday'
              when 3 then 'Tuesday'
              when 4 then 'Wednesday'
              when 5 then 'Thursday'
              when 6 then 'Friday'
              else 'Weekend'
       end as [Day]
FROM dbo.Sales s;
```

Additional Common Built-In Functions

The query below uses the **ISNULL** function to replace NULL values in the MiddleInitial column with an empty space. The **REPLACE** function replaces two adjacent space bar characters with an empty space. This way, if a row contains a NULL value for MiddleInitial, there won't be any extra spaces when FirstName, MiddleInitial, and LastName are concatenated.

```
SELECT
       PersonID
       ,FirstName
       ,LastName
       ,MiddleInitial
       ,LastName + ', ' + FirstName as [Full Name]
       ,REPLACE(FirstName + ' ' + ISNULL(MiddleInitial,'') + ' ' + LastName,'  ','') as
[Name with MI]
FROM dbo.Person;
```

In the function below the **LEFT** function returns the first character from the left side of the FirstName column. The **RTRIM** function shaves any blank spaces from the right side of the string. LEFT and RTRIM are string functions, and only work with string data types.

```
SELECT
       PersonID
       ,FirstName
       ,MiddleInitial
       ,LastName
       ,RTRIM(LEFT(FirstName,1) + ISNULL(MiddleInitial,'')) + LEFT(LastName,1)
FROM dbo.Person;
```

The query below demonstrates the **ABS**, **CEILING**, and **SQRT** functions.

```
SELECT
	SaleID
	,ABS(0 - SalesAmount) AS [Absolute value]
	,CEILING(SalesAmount) AS [Rounded up]
	,SQRT(SalesAmount) AS [Square root]
FROM dbo.Sales
```

The query below demonstrates the **SUBSTRING**, **REVERSE**, **LEN**, and **UPPER** functions.

```
SELECT
	PersonID
	,LastName + ', ' + FirstName AS [Full name]
	,SUBSTRING(LastName + ', ' + FirstName,1,5) AS [First five characters]
	,REVERSE(LastName + ', ' + FirstName) AS [Full name in reverse]
	,LEN(LastName + ', ' + FirstName) AS [Number of characters]
	,UPPER(LastName + ', ' + FirstName) AS [Upper case]
FROM dbo.Person;
```

A complete list and description of all of SQL Server's built-in functions can be found here (https://msdn.microsoft.com/en-us/library/ms174318.aspx).

Table Aliases

In the examples we've seen thus far, we've only listed one table in the FROM clause, thus the query processor can assume each column in the SELECT list can be found in the single table in the FROM clause. However, we can explicitly identify the table by prefacing the column names in the SELECT list with the table name, as demonstrated in the following query.

```
SELECT
	dbo.Cars.CarID
	,dbo.Cars.Make
	,dbo.Cars.Model
	,dbo.Cars.Year
FROM dbo.Cars;
```

As a bit of a shortcut, we can give the table name an **alias** in the FROM clause. As the FROM clause is processed first, this alias becomes the way we'll refer to the table in the rest of the query. This is demonstrated in the query below.

```
SELECT
	c.CarID
	,c.Make
	,c.Model
	,c.Year
FROM dbo.Cars as c;
```

Or we can leave out "as", and make it just…

```
SELECT
	c.CarID
```

```
        ,c.Make
        ,c.Model
        ,c.Year
FROM dbo.Cars c;
```

At this point you may be wondering why anyone would use a table alias. In a later module we'll begin using multiple tables in the FROM clause, and table aliases will make our code much easier to read.

WHERE clause

Thus far we've seen how to retrieve entire columns of data from a table. But what if we want to filter the results by applying a conditional expression that identifies specific records to return? This is where the **WHERE** clause comes in. In the query below we're limiting our results to just those records with a Year value greater than 1969.

```
SELECT
        c.CarID
        ,c.Make
        ,c.Model
        ,c.Year
FROM dbo.Cars c
WHERE c.Year > 1969;
```

Here's what happens when you execute this. First, the FROM clause is processed and the query processor retrieves all the records from the dbo.Cars table. Next, the WHERE clause is processed. The conditional expression (c.Year > 1969) is called a ***predicate***. It determines which rows from dbo.Cars table are actually included in the result set. The predicate can contain any valid logical expression that evaluates to true, false, or null. Finally, the SELECT list is processed, and data in the listed columns is returned.

The predicate is simply a logical expression that gets applied to each row, and returns a value of true, false, or null for the row. If the value is true, the row is returned, if false or null, it isn't returned.

Here's another example showing a more complex predicate. In this example, we test each row to see if the Year column value is greater than 1969 and less than 1975, or if the DealerPrice is greater than 10000. If either of these logical conditions is true, then the predicate evaluates to true, and the record is included in the result set.

```
SELECT
        c.CarID
        ,c.Make
        ,c.Model
        ,c.Year
FROM dbo.Cars c
WHERE (c.Year > 1969 AND c.Year < 1975 ) OR DealerPrice > 10000;
```

Another operator that is often used in WHERE clauses is the **IN** operator. It tries to match a column value to items in a list, and if a match is found evaluates to true. This is equivalent to using two OR operators. The two queries below are functionally equivalent.

```
SELECT
	c.CarID
	,c.Make
	,c.Model
	,c.Year
FROM dbo.Cars c
WHERE Make IN ('Pontiac','Mercury','Dodge');

SELECT
	c.CarID
	,c.Make
	,c.Model
	,c.Year
FROM dbo.Cars c
WHERE Make ='Pontiac' OR Make = 'Mercury' OR Make='Dodge';
```

The following query demonstrates use of the **LIKE** operator. The LIKE operator is used for string comparisons. The query below will return all Person records with a LastName value starting with the letter A. The % sign acts as a wildcard. Additional pattern matching operations available with LIKE can be found here (https://msdn.microsoft.com/en-us/library/ms179859.aspx).

```
SELECT
	p.FirstName
	,p.LastName
	,p.Email
FROM dbo.Person p
WHERE p.LastName LIKE 'A%';
```

DISTINCT

Sometimes you want to retrieve a distinct list of rows from a table with many duplicate column values. For example, if we look at the Cars table, we've got many rows with the same value for Make and Model. The following query will return 2407 rows, for example.

```
SELECT
	c.Make
FROM Cars c;
```

If we look carefully at the result set returned by this query, we'll see many duplicate values. By including keyword **DISTINCT**, we eliminate all duplicates and return 12 rows, with a distinct value for Make on each row.

```
SELECT DISTINCT
	c.Make
FROM Cars c;
```

DISTINCT removes duplicate rows, so the following query will return the 30 distinct combinations of MAKE and MODEL.

```
SELECT DISTINCT
       c.Make
       ,c.Model
FROM CARS c;
```

ORDER BY

Often you'll want the retrieved data set to be sorted.. For example, you may want to sort a list of sales records by the sales date. The **ORDER BY** clause allows you to sort records. Without ORDER BY, there is no specific sort order guaranteed in the result set.

The following query will return the SaleID, CarID, SalesAmount, and SalesDate columns from the dbo.Sales table sorted by SalesDate in ascending order (earliest to most recent):

```
SELECT
       SaleID
       ,CarID
       ,SalesAmount
       ,SalesDate
FROM dbo.Sales
ORDER BY SalesDate;
```

Ascending order is the default, but can be specified explicitly with **ASC**. Descending order can be specified with **DESC**. The following query sorts all sales records first by date in ascending order, and then by sales amount in descending order.

```
SELECT
       SaleID
       ,CarID
       ,SalesAmount
       ,SalesDate
FROM dbo.Sales
ORDER BY SalesDate ASC, SalesAmount DESC;
```

In a relational database, the results of a SELECT query cannot be assumed to be in any particular order except the order specified in an ORDER BY clause. Without ORDER BY, you cannot assume sorted results, even if the results appear to be sorted.

Also, because of the existence of indexes, the database is usually the most efficient place to perform sort operations. Therefore, when assorted data set is needed, it's almost always going to be best to use an ORDER BY in your SELECT queries to perform sort operations rather than performing the sort within the client application.

TOP

Let's say you're interested in finding the top 10 sales in the table dbo.Sales. As we discussed above we can use ORDER BY to sort the records by sale from highest to lowest as follows:

```
SELECT
       SaleID
       ,CarID
       ,SalesAmount
       ,SalesDate
FROM dbo.Sales
ORDER BY SalesAmount DESC;
```

However, what if we want just the top 10 sales? In this case, we can use keyword TOP. For example, the following would return the 10 highest sales:

```
SELECT TOP 10
       SaleID
       ,CarID
       ,SalesAmount
       ,SalesDate
FROM dbo.Sales
ORDER BY SalesAmount DESC;
```

The following would return the top 10 percent:

```
SELECT TOP 10 Percent
       SaleID
       ,CarID
       ,SalesAmount
       ,SalesDate
FROM dbo.Sales
ORDER BY SalesAmount DESC;
```

Summary

The query below demonstrates the use of column and table aliases, scalar functions, the WHERE clause, and ORDER BY.

```
SELECT
        s.SaleID AS [SaleID]
       ,s.CarID AS [Car]
       ,s.CustomerID AS [Customer]
       ,s.SalesPersonID AS [Sales Person]
       ,YEAR(s.SalesDate) AS [Year of Sale]
       ,CEILING(s.SalesAmount * .15) AS [Sales Commission]
       ,s.SalesAmount As [Amount of Sale]
FROM dbo.Sales s
WHERE s.SalesDate > '2009-01-01'
ORDER BY s.SalesAmount DESC;
```

The query below demonstrates the use of column and table aliases, scalar functions, and the LIKE operator in the WHERE clause.

```
SELECT
       PersonID
       ,UPPER(p.LastName + ', ' + p.FirstName) AS [Full Name]
```

```
      ,UPPER(City + ', ' + State + ' ' + Zip) AS [Location]
FROM dbo.Person p
WHERE p.LastName LIKE 'S%'
ORDER BY p.LastName, p.FirstName;
```

Additional Reading

https://technet.microsoft.com/en-us/library/ms174318.aspx

(https://msdn.microsoft.com/en-us/library/ms190309.aspx

https://technet.microsoft.com/en-us/library/ms187928.aspx

https://www.microsoft.com/en-us/download/details.aspx?id=35834

https://msdn.microsoft.com/en-us/library/ms186724.aspx

https://msdn.microsoft.com/en-us/library/ms179859.aspx