`SELECT` statement consist of 4 components

```
1. `Select` – Required
2. `From` – Required
3. `Where` – Optional
4. `Order By` – Optional
```

```
SELECT [Name]
FROM [HumanResources].[Department]
WHERE [Name] = 'Production'
ORDER BY [Name]
```

Schemas

A name collection of database objects that form a namespace
Eg. `Person` and `HumanResources` are schemas.

```
Person.Address
Person.BusinessEntity

HumanResources.JobCandidate
HumanResources.Shift
```

A schema is also a secure object so a D-B can grant explicit permissions to run those schemas

But `dbo.Customer` - `dbo` -> is a owner

Four parts naming conversion

1 2 3 4
`SELECT AdventureWorks2014.HumanResources.JobCandidate.Resume FROM HumanResources.JobCandidate`

Aliasing

Giving another name to a column

`ProductionColumn` - Alias

```
SELECT [Name] as ProductionColumn
FROM [HumanResources].[Department]
WHERE [Name] = 'Production'
ORDER BY [Name]
```

Derived Column

A column that doesn't exist until we define one within the confines of our `SELECT` statement
`Purchase` - Derived Column
here, * is multiply not 'select all'

```
SELECT UnitPrice * OrderQty as Purchase
FROM Sales.SalesOrderDetail
```

DATEDIFF *function*

Using the `DATEDIFF` function to get the ages of employees.

`YY` - Interested in the years
`BirthDate` - Starting date
`GetDate()` - Ending date
`Age` - Alias

```
SELECT BirthDate,
        DATEDIFF(YY, BirthDate, GetDate()) as Age
FROM    HumanResources.Employee
```

View

An optimized table-like, select-query-like object that is optimized and stored in the database but doesn't store any data.

Views are often used to hide data. An organization can use view to restrict what the end user will see.

`vwReturnAllRows` - View

```
CREATE VIEW vwReturnAllRows
AS
SELECT * FROM Customer
GO

SELECT * FROM vwReturnAllRows

/*
Output:
CustomerID  Lastname    Firstname   Email
1           Sheen       Charlie     sheen@example.com

2           John        Doe         johndoe@abc.com

*/
```

Alter views

```
ALTER VIEW vwReturnAllRows
AS
SELECT Lastname, Firstname FROM Customer
GO

SELECT * FROM vwReturnAllRows

/*
Output:
Lastname    Firstname
Sheen       Charlie
John        Doe

*/
```

`SELECT * FROM vwReturnAllRows` returns only `Lastname` and `Firstname` and the `Email` and `CustomerID` columns have been hidden from the user.

Operators

Examples: `=` `>=`

`<>` -Not equal to

```
SELECT Name, ListPrice
FROM    Production.Product
WHERE ListPrice > 100
ORDER BY ListPrice desc
```

LIKE

Like doesn't mean 'similar' in MSSQL, it means 'exactly like'. Works on both numbers and letters

```
SELECT ListPrice
FROM Production.Product
WHERE ListPrice LIKE '20%' -- Where ListPrice that starts with '20'

SELECT ListPrice
FROM Production.Product
WHERE ListPrice LIKE '%64%' -- Where ListPrice contains '64'

SELECT ListPrice
FROM    Production.Product
WHERE ListPrice LIKE '_09%' -- Where ListPrice has 0 in the 2nd and 9 in 3rd place

SELECT ListPrice
FROM Production.Product
WHERE ListPrice LIKE '2______%' -- Where ListPrice starts with 2 and has at least 7
chars in length. Each _ represents a char, so 6(_)'s + 2 = 7 chars
    -- You can seperate the underscores with '%' . Example below returns the same
result
SELECT ListPrice
FROM Production.Product
WHERE ListPrice LIKE '2_%_%_%_%_%_%'


SELECT * FROM HumanResources.Department
WHERE Name LIKE '%E' -- Where Name ends with 'E'
```

Logical Comparisons

`AND` `OR` `NOT`

```
SELECT * FROM HumanResources.Employee
WHERE MaritalStatus = 'M' AND VacationHours > 25

SELECT * FROM HumanResources.Employee
WHERE MaritalStatus = 'M' OR VacationHours > 25

SELECT * FROM HumanResources.Employee
WHERE NOT SalariedFlag = 1
```

NULL

`IS NULL`

`NULL` is not equal to `NULL`

```

```

```
SELECT * FROM Production.Product
WHERE Color IS NULL

SELECT * FROM Production.Product
WHERE Color IS NOT NULL

/*
Output: No Results
In SQL, NULL means nothing and can't be compared to anything
SO Title = NULL returns nothing.
*/
SELECT FirstName
FROM Person.Person
WHERE Title = NULL AND BusinessEntityID < 7
```

The SQL Server Database Engine uses a bitmap to track which columns in a row are null and which are not. The Bitmap contains a bit for each column with the bit set to 1 if the column is null, i.e, if the value is missing

Extended Filtering *using expressions*

```
SELECT NationalIDNumber, LoginID
FROM HumanResources.Employee
WHERE BirthDate >= '1962-1-1' AND BirthDate <= '1985-12-31' -- Filtering using
Expressions

-- Same as above using Between
SELECT NationalIDNumber, LoginID
FROM HumanResources.Employee
WHERE BirthDate BETWEEN '1962-1-1' AND '1985-12-31'


SELECT *
FROM Production.Product
WHERE ProductSubcategoryID IN (1,2,3) -- Where ID is in the list of (1,2,3)

-- Using IN in Sub-queries
SELECT * FROM Production.Product
WHERE ProductSubcategoryID IN
    (
        SELECT ProductSubcategoryID
        FROM Production.ProductSubcategory
        WHERE ProductCategoryID IN (1,2)
    )
```

Order of Operator processing

1. `NOT` 2. `AND` 3. `OR`

Filtering and Grouping Operations using Parenthesis

```sql
/*
In this case, it doesn't really matter since AND is processed before OR anyway but
in some cases, it may matter
*/
SELECT Name,
    ProductNumber,
    ListPrice,
    ProductSubCategoryID
FROM Production.Product
WHERE (ProductSubcategoryID = 1 AND ListPrice > 100)
OR    (ProductSubcategoryID = 2 AND ListPrice > 500)
```

Variables

Placeholders for values. Declared using `DECLARE` and assigned a value using `SET`.
`@` is placed in front of the variable name to declare it as a variable.

```sql
DECLARE @MyNumber
INT SET @MyNumber = 1
SELECT @MyNumber
```

Variables can also be assigned a type when declared. A `SELECT` statement can also be used to assign a value to a variable.

```sql
DECLARE @var1 nvarchar(30);
SELECT @var1 = 'Generic Name'; -- default name when ID is not found
SELECT @var1 = Name
FROM Sales.Store
WHERE BusinessEntityID = 1000;
SELECT @var1 AS 'Company Name' -- Column name as 'Company Name'
```

Function

Purpose of a function is to return a value. Most functions return a scalar value (a single unit of data). Functions can return any data type. Functions are divided into two different groups based on **determinism**.

Determinism is whether the outcome of a function can be predicted based on it's input parameters or by executing at one time.

If a function is not dependent on any external factors other than the value of input parameters, it is said to be **deterministic**.
If the output can vary based on any conditions in the environment or algorithms that produce random or dependent variables, then the function is **non-deterministic**.

more or less like pure and impure functions in functional programming

Eg. `GETDATE()` is non-deterministic because it will never return the same value.

Combining *functions* with query expressions to modify column values

```sql
SELECT JobTitle, NationalIDNumber, YEAR(BirthDate) AS BirthYear
FROM HumanResources.Employee
```

Nested Functions

```sql
SELECT CONVERT(VARCHAR(20), GETDATE(), 101)
```

User-defined Functions

```
CREATE FUNCTION fx_SumTwoValues
    (@Val1 INT, @Val2 INT)
RETURNS INT

AS

BEGIN
RETURN (@Val1 + @Val2)
END

SELECT dbo.fx_SumTwoValues(1,4) AS SumOfTwoValues
```

Inline Table-valued Functions (no `BEGIN` and `END`)

```
CREATE FUNCTION tbfReturnFirstName(@lastName nvarchar(15))
RETURNS TABLE
AS
RETURN
    (SELECT Firstname FROM Customer WHERE Lastname = @lastName)
GO

SELECT * FROM tbfReturnFirstName('Adams')
```

Multi-Statement Table Valued User Defined Function

A user-defined function that returns a table. Can have one or more than one SQL statement.

```
-- Creates a table and fills it with 1000 values
CREATE TABLE Table1
    (ID INT, VALUES_INSERT NVARCHAR(100))

DECLARE @count INT
SET @count = 0
WHILE @count < 1000
BEGIN
    INSERT INTO Table1
    SELECT @count, @count
SET @count = @count + 1
END
GO

-- Create a function called 'estimatedRows'
-- Just like the table above, but a function
CREATE FUNCTION dbo.estimatedRows(@ID INT)
RETURNS @IDS TABLE
    (ID INT NOT NULL, VALUES_INSERT NVARCHAR(100) NOT NULL)
AS
BEGIN
    INSERT INTO @IDS
    SELECT ID,VALUES_INSERT FROM Table1
    WHERE ID > @ID
RETURN
END
GO

SELECT * FROM dbo.estimatedRows(0)
```

Aggregate Functions

Return single values from query statement. Eg. What is the average sales for a particular month.

```
USE AdventureWorks2014

SELECT MIN(ModifiedDate) FROM Sales.Store

SELECT MAX(ModifiedDate) FROM Sales.Store

SELECT MIN(SalesPersonID) FROM Sales.Store

SELECT MAX(SalesPersonID) FROM Sales.Store

SELECT AVG(SalesPersonID) FROM Sales.Store

SELECT SUM(SalesPersonID) FROM Sales.Store
```

COUNT(*) and COUNT(Distinct)

`Count *` returns the count of all the available rows.

`Count(Distinct)` returns the count of distinct(unique) values (counting duplicates as one) in that column

```
SELECT COUNT(SalesPersonID) AS CountResult FROM Sales.Store

SELECT COUNT(Distinct SalesPersonID) AS CountResult FROM Sales.Store
```

Conversion or Cast Functions

Functions that 'convert' the type of one value to the other

```
SELECT CAST('123' AS INT)
SELECT CAST('123.4' AS FLOAT)
SELECT CAST('123.4' AS DECIMAL(9,2)) -- 9 - Precision => num of values that can be
stored to both left and right the left. 2 - Scale of digits that can be stored on
the right of the decimal point

SELECT 'Current Datetime: ' + CONVERT(VARCHAR(50), GETDATE(), 100)
SELECT 'US Date: ' + CONVERT(VARCHAR(50), GETDATE(), 101)
```

Sub Query

A `select` query within a `select` query. Embedded select statements can be used to return a single column value also known as **Scalar Value Expression**

```
SELECT ProductID,
        UnitPrice,
        (SELECT AVG(UnitPrice) FROM Sales.SalesOrderDetail) AS AvgPrice -- Scalar
Value Expression
FROM Sales.SalesOrderDetail


SELECT ProductID,
        UnitPrice - (SELECT AVG(UnitPrice) FROM Sales.SalesOrderDetail) AS
AvgPriceDiff
FROM Sales.SalesOrderDetail
```

Having Clause

Involves the use of `Having`. Functions like the `WHERE` clause but used after the aggregate function whereas `WHERE` is used before

```
SELECT Name from Production.Product
WHERE EXISTS
    (SELECT SUM(UnitPrice) FROM Sales.SalesOrderDetail
    WHERE SalesOrderDetail.ProductID = Product.ProductID
    HAVING SUM(UnitPrice) > 2000
    )
```

Common Table Expression (CTE)

A sub-query that only exists in memory so doesn't require special permissions or necessary physical disk operations. It's a named object that can be reused and referenced just like a table. Useful for performance tuning . It allows us to skip the *System Database*. It doesn't 'slam' `tempdb` in *System Databases* because it just lives in memory. Starts with a `WITH`

```
WITH cteAllCustomers
AS
(SELECT * FROM Customer)
SELECT * FROM Customer


USE AdventureWorks2014
GO

SELECT P.* FROM (
    SELECT ProductID, Name, ListPrice
    FROM Production.Product WHERE ListPrice > 0) AS P

-- The above query re-written as a CTE
WITH cteNonFreeProducts (ProductID, Name, ListPrice)
AS
    (SELECT ProductID, Name, ListPrice
    FROM Production.Product
    WHERE ListPrice > 0)
    SELECT * FROM cteNonFreeProducts
```