

# Module 2:

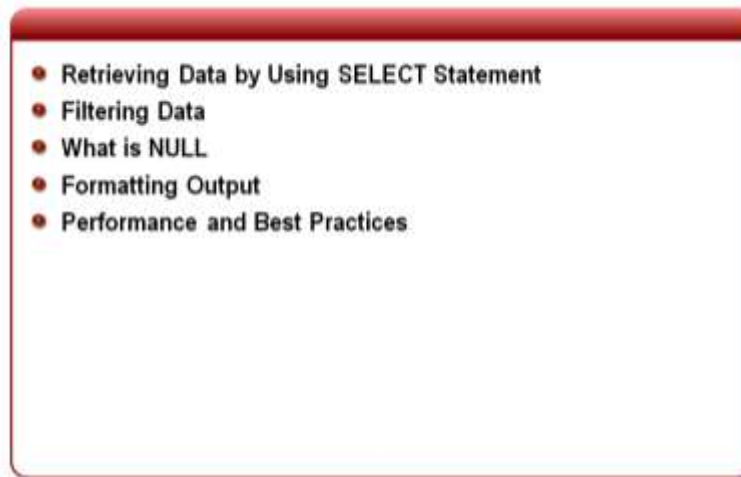
## Retrieving and Filtering Data

### Contents

<b>Module Overview .....</b>	<b>1</b>
<b>Lesson 1: Retrieving Data by Using SELECT Statement.....</b>	<b>2</b>
Elements of SELECT Statement .....	3
Retrieving Columns and Data from a Table .....	4
Working with String Functions.....	6
Practice: Retrieving Data by using SELECT statement.....	8
<b>Lesson 2: Filtering Data .....</b>	<b>10</b>
Selecting Specific Rows from a Table .....	11
Comparison Operators .....	12
String Comparisons .....	13
Logical Operators .....	16
Retrieving a Range of Values.....	18
Retrieving a List of Values .....	19
Practice: Filtering Data .....	20
<b>Lesson 3: What is NULL .....</b>	<b>22</b>
Definition of NULL .....	23
Working with NULL Values .....	24
NULL Value Practice by Example .....	25
Practice: Working with NULL.....	27
<b>Lesson 4: Formatting Output .....</b>	<b>29</b>
Sorting Data (ORDER BY) .....	30
What is DISTINCT? .....	32
Labeling Columns .....	34
Formatting Data with String Literals .....	36
Using Expressions.....	37
Practice: Formatting the Query Output .....	38
<b>Lesson 5: Performance and Best Practices .....</b>	<b>40</b>
Basics About Performance Issues.....	41
Recommended Practices.....	42
Executions and Query Tuning.....	43
Practice: Query Tuning .....	44
<b>Summary .....</b>	<b>47</b>



# Module Overview



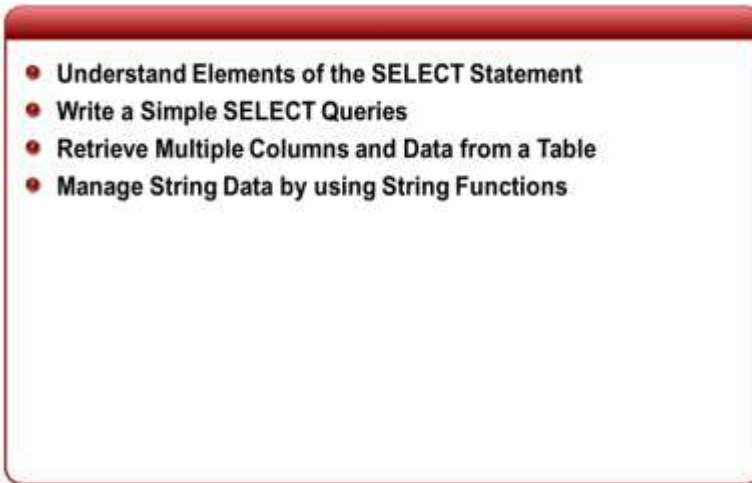
Databases are one of the cornerstones of modern business companies. The idea of storing data inside of the database has arisen from the necessity to access and retrieve those data at any moment. Data retrieval is usually made with SELECT statement and is therefore very important that in this module you master its use. Retrieved data is often not organized in the way you want them to be, so they require additional formatting. Besides formatting, accessing very large amount of data requires you to take into account the speed and manner of query execution which can have a major impact on system performance.

## Objectives

After completing this module, you will be able to:

- Retrieve data by using SELECT statement
- Filter data by using WHERE clause
- Work with NULL values
- Format data output
- Consider impact of query execution on performance

# Lesson 1: Retrieving Data by Using SELECT Statement



Databases are usually consisted of several tables where all data are stored. For example, the Adventure Works database that you will use in this book contains 71 tables such as Customers, Products, and Orders etc. Table names clearly describe entities whose data are stored inside and therefore if you need to create a list of new products or a list of customers who had the most orders, you need to retrieve those data by creating a query. A query is an inquiry into the database by using the SELECT statements which is first and fundamental SQL statement that we are going to introduce in this book.

## Objectives

After completing this lesson, you will be able to:

- Understand the elements of the SELECT statement
- Write a simple SELECT queries
- Retrieve multiple columns and data from a table
- Work with string functions

## Elements of SELECT Statement

- **SELECT statement consists of a set of clauses:**
  - **INTO** - enables you to insert data (retrieved by the SELECT clause) into a different table
  - **FROM** - specifies the source of the data
    - **FROM clause is mandatory** in SELECT statement
  - **WHERE** - specifies search conditions
  - **ORDER BY** - orders query results
  - **GROUP BY** - arranges identical data into groups
  - **HAVING** -allows creation of selection criteria at the group level

The SELECT statement is the basis for constructing queries which enables you to retrieve data from the database table. SELECT statement consists of a set of clauses that specifies which data will be included into query result set. All clauses of SQL statements are the keywords and because of that will be written in capital letters.

Syntactically correct SELECT statement requires a mandatory FROM clause which specifies the source of the data you want to retrieve. Besides mandatory clauses there are a few optional ones that can be used to filter and organize data:

- The INTO clause enables you to insert data (retrieved by the SELECT clause) into a different table. It is mostly used to create table backup.
- The WHERE clause places conditions on a query and eliminates rows that would be returned by a query without any conditions.
- The ORDER BY clause displays the query result in either ascending or descending alphabetical order.
- The GROUP BY clause provides mechanism for arranging identical data into groups.
- The HAVING clause allows you to create selection criteria at the group level.


## Retrieving Columns and Data from a Table

- **SELECT clause is followed by a list of comma separated column names**

```
SELECT Name, ProductNumber
FROM Production.Product
```

- **It is good practice to use both schema (Production) and object name of the table (Product).**
- **To retrieve all columns use wildcard character (\*)**

```
SELECT *
FROM Production.Product
```



- **Avoid (\*) unless you really need to retrieve all columns.**

The SELECT clause in a query is followed by a list of comma separated column names that you wish to retrieve in a result set. The following code sample retrieves data from columns Name and ProductNumber stored inside the Product table.

```
SELECT Name, ProductNumber
FROM Production.Product
```

Result is:

Name	ProductNumber
Adjustable Race	AR-5381
Bearing Ball	BA-8327
BB Ball Bearing	BE-2349
Headset Ball Bearings	BE-2908
Blade	BL-2036
....	
ML Bottom Bracket	BB-8107
HL Bottom Bracket	BB-9108
Road-750 Black, 44	BK-R19B-44
Road-750 Black, 48	BK-R19B-48
Road-750 Black, 52	BK-R19B-52

(504 row(s) affected)

Each object (table, view, etc.) in the database belongs to a particular scheme. At the scheme level, object names are unique and therefore when specifying data source it is considered as a good practice to use both scheme (Production) and object name of the table (Product).

As a result of the query execution, we received 504 entries that include only data on product name and number. In order to retrieve all columns from Production.Product table you can use wildcard character (\*) that means *all columns*.

```
SELECT *
FROM Production.Product
```

However, especially when working with a production database, `SELECT *` should be avoided unless you really need to retrieve all columns. Depending on the amount of retrieved data usage of the wildcard, character can cause not only server and network performance reduction but also a result set that is difficult to read and analyze.

## Working with String Functions

• Most commonly used string functions are:

Function	Example	Description
<b>SUBSTRING</b>	<code>SUBSTRING (expression, start, length)</code>	Returns part of a expression passed in as an argument
<b>LEFT, RIGHT</b>	<code>LEFT (expression, integer_Value)</code> <code>RIGHT(expression, integer_Value)</code>	Returns the specified number of characters from the specified side (left or right) of the expression
<b>UPPER, LOWER</b>	<code>UPPER(expression)</code> <code>LOWER(expression)</code>	Returns uppercase or lowercase version of all characters in the expression
<b>REPLACE</b>	<code>REPLACE (string_expression, search_string, replacement_string)</code>	Replaces occurrence of the string which specified as the search_string with replacement_string
<b>LEN, DATALENGTH</b>	<code>LEN (string_expression)</code> <code>DATALENGTH(expression)</code>	LEN function returns the length (number of characters) of a string expression, DATALENGTH returns the number of bytes used to represent any expression

Regardless of the environment for which they provide support, it is common for databases to contain considerable amount of string data which often requires some form of manipulation. String data manipulation is mostly used in cases when it is necessary to represent strings in format which is different from the one stored in the table, for example to extract substrings, change letter case, etc. The easiest ways to manipulate string data is to use string functions that take character string as input and produce another character string as output. Some of the most commonly used string functions are:

- **SUBSTRING** - Returns part of an expression passed in as an argument. The following example extracts substring from expression Headset Ball Bearings that starts from the ninth character and is four characters long.

```
SELECT SUBSTRING ('Headset Ball Bearings', 9, 4)
```

Result is:

```
-----
Ball
```

(1 row(s) affected)

- **LEFT, RIGHT** - Returns the specified number of characters from one side (left or right) of the expression

```
SELECT LEFT ('Headset Ball Bearings', 7)
```

Result is:

```
-----
Headset
```

(1 row(s) affected)



- **UPPER, LOWER** - Returns uppercase or lowercase version of all characters in the expression

```
SELECT LOWER('Headset Ball Bearings')
```

Result is:

```
-----
headset ball bearings
(1 row(s) affected)
```

- **REPLACE** - Replaces occurrence of the string specified as the search\_string (Ball) with replacement\_string (BALL)

```
SELECT REPLACE('Headset Ball Bearings', 'Ball', 'BALL')
```

Result is:

```
-----
Headset BALL Bearings
(1 row(s) affected)
```

- **LEN, DATALENGTH** - LEN function returns the length (number of characters) of a string expression while DATALENGTH returns the number of bytes used to represent any expression

```
SELECT LEN('Headset Ball Bearings')
```

Result is:

```
-----
21
(1 row(s) affected)
```

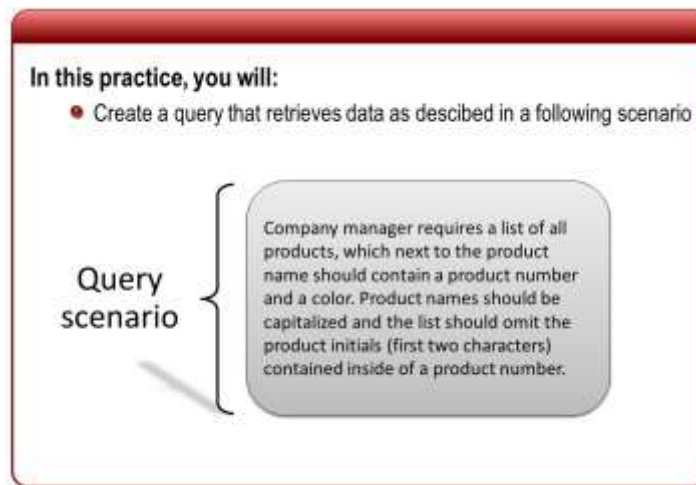
We will mention also a CONCAT function which can be used to combine together (concatenate) the results from several different columns.

```
SELECT CONCAT(ProductNumber, ', ', Color, ', ', ListPrice)
FROM Production.Product
```

Result is:

```
-----
AR-5381, , 0.00
BA-8327, , 0.00
....
BK-R19B-44, Black, 539.99
BK-R19B-48, Black, 539.99
BK-R19B-52, Black, 539.99
(504 row(s) affected)
```

## Practice: Retrieving Data by using SELECT statement



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 1 from Module 2.

To successfully complete the exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - <http://msftdbprodsamples.codeplex.com/>

### Exercise 1: Retrieving Data by using SELECT statement

1. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
2. On the Connect to Server dialog windows, under Server name type the name of your local instance.
  - a. If it is default then just type (local)
3. Use Windows Authentication
4. Open File menu →New →Database Engine Query

#### *Query Scenario*

Company manager requires a list of all products, which next to the product name should contain a product number and color. Product names should all be capitalized and the list should omit the product initials (first two characters) contained inside of a product number.

5. Type following TSQL code in query windows:

```

SELECT  UPPER (Name) ,
        SUBSTRING (ProductNumber , 4 , 6) ,
        Color
FROM Production.Product

```

6. Click on Execute or press F5 to run

Result is:

		Color
-----		-----
ADJUSTABLE RACE	5381	NULL
BEARING BALL	8327	NULL
BB BALL BEARING	2349	NULL
HEADSET BALL BEARINGS	2908	NULL
BLADE	2036	NULL
....		
HL BOTTOM BRACKET	9108	NULL
ROAD-750 BLACK, 44	R19B-4	Black
ROAD-750 BLACK, 48	R19B-4	Black
ROAD-750 BLACK, 52	R19B-5	Black

(504 row(s) affected)

Modify the previous query in a way that the product number displays product initials but the middle part of the label (four digit ones which can contain characters) should be replaced with X characters.

6. Type following TSQL code in query windows:

```

SELECT Name ,
        REPLACE (ProductNumber , SUBSTRING (ProductNumber , 4 , 4) , 'XXXX' )
FROM Production.Product

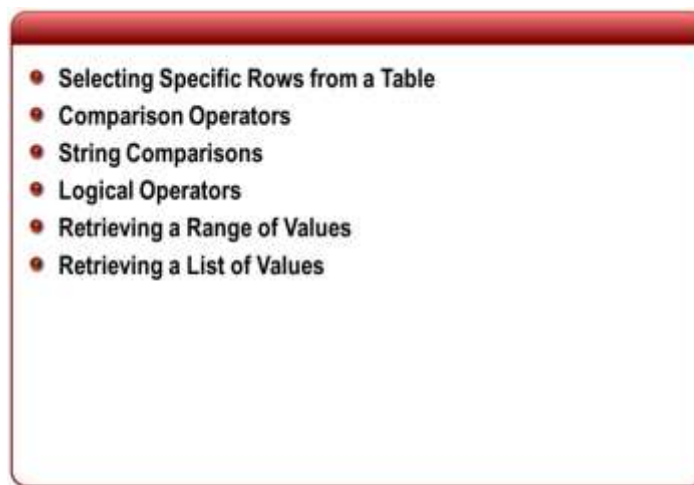
```

Result is:

Name	
-----	-----
Adjustable Race	AR-XXXX
Bearing Ball	BA-XXXX
BB Ball Bearing	BE-XXXX
....	
LL Bottom Bracket	BB-XXXX
ML Bottom Bracket	BB-XXXX
HL Bottom Bracket	BB-XXXX
Road-750 Black, 44	BK-XXXX-44
Road-750 Black, 48	BK-XXXX-48
Road-750 Black, 52	BK-XXXX-52

(504 row(s) affected)

## Lesson 2: Filtering Data



In practice, there are very few cases where you want to show all the data contained in the table. Therefore, most of the time you will need data that meet certain conditions or, in other words, you will need filtered data. By using the **WHERE** clause in the **SELECT** statement you can specify search conditions and return only those rows (record or records) that meet specific criteria.

### Objectives

After completing this lesson, you will be able to:

- Filter data by using **WHERE** clause
- Filter data by using comparison operators
- Filter data by using string comparisons
- Filter data by using logical operators
- Retrieve a range of values
- Retrieve a list of values

## Selecting Specific Rows from a Table

- **WHERE clause enables you to retrieve only those data that fulfill certain conditions**

```
SELECT Name, ProductNumber, Color
FROM Production.Product
WHERE DaysToManufacture = 2
```

- **Only rows that match condition (can be manufactured in two days) will be retrieved as a part of the result set**

Name	ProductNumber	Color
HL Mountain Frame - Black, 38	FR-M94B-38	Black
HL Mountain Frame - Silver, 38	FR-M94S-38	Silver
ML Mountain Frame - Black, 38	FR-M63B-38	Black
ML Road Frame-W - Yellow, 38	FR-R72Y-38	Yellow
ML Mountain Frame-W - Silver, 38	FR-M63S-38	Silver
LL Mountain Frame - Black, 40	FR-M21B-40	Black
LL Mountain Frame - Silver, 40	FR-M21S-40	Silver

(7 row(s) affected)

By using WHERE clause of the SELECT statement you are able to retrieve only those data that fulfill certain conditions. The conditions that are specified in the WHERE clauses are known as predicates and in most cases they result with one of the Boolean values TRUE or FALSE. However, predicates can result with an UNKNOWN value which will be discussed in the following lessons.

Let's look at the example which creates a list of products that can be manufactured in two days. Number of days necessary to manufacture some product is stored as an integer value in column DaysToManufacture.

```
SELECT Name, ProductNumber, Color
FROM Production.Product
WHERE DaysToManufacture = 2
```

Result is:

Name	ProductNumber	Color
HL Mountain Frame - Black, 38	FR-M94B-38	Black
HL Mountain Frame - Silver, 38	FR-M94S-38	Silver
ML Mountain Frame - Black, 38	FR-M63B-38	Black
ML Road Frame-W - Yellow, 38	FR-R72Y-38	Yellow
ML Mountain Frame-W - Silver, 38	FR-M63S-38	Silver
LL Mountain Frame - Black, 40	FR-M21B-40	Black
LL Mountain Frame - Silver, 40	FR-M21S-40	Silver

(7 row(s) affected)

Only rows for which predicate evaluates to TRUE will be returned as a part of the result set (only 7 rows matched the specified condition).

## Comparison Operators

• Comparison operators are used to compare data with a specific value or expression

Operator	Description
=	equals
<>, !=	does not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

```
SELECT Name, Weight
FROM Production.Product
WHERE Weight >= 1000
```

Comparison operators are used when it is necessary to compare data from the table with a specific value or expression. It is important to note that you can compare only compatible values which are defined by a data type. This means that you will not be able to compare the string with decimal values. The following is a list of the basic comparison operators that can be used in the WHERE clause:

Operator	Description
=	equals
<>, !=	does not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

The next example returns a list of products whose weight is greater than or equal to 1000.

```
SELECT Name, Weight
FROM Production.Product
WHERE Weight >= 1000
```

Result is:

Name	Weight
LL Road Rear Wheel	1050.00
ML Road Rear Wheel	1000.00

(2 row(s) affected)

## String Comparisons

- Basic comparison can be made by using equal operator (=)
- For more advanced forms of comparisons you can use one of the following functions:

Function	Description
LIKE	searches for a partial match
CONTAINS	searches for exact or literal matches at the word level
FREETEXT	searches for expanded matches at the word level

```
SELECT Name, ProductNumber
FROM Production.Product
WHERE Name LIKE '[^A]_F%'
```

In addition to the basic operators described above, T-SQL also supports comparison operators for evaluating string data types. The most basic form of string comparison can be made with equal (=) operator as shown in the next example which retrieves Name and ProductNumber of the product named Chainring.

```
SELECT Name, ProductNumber
FROM Production.Product
WHERE Name = 'Chainring'
```

Result is:

Name	ProductNumber
Chainring	CR-7833

(1 row(s) affected)

In order to create more complex comparisons you will need to use some of the advanced mechanisms such as:

- **LIKE** – is used in cases when search criteria are only partially known. In order to specify the missing parts of the value you can use one of the following wildcard characters:

Wildcard character	Description	Usage
<b>% (percent)</b>	replaces zero or more characters	<b>LIKE 'Ch%'</b> will match Chain, Chain Stays, Chainring or any string starting with 'Ch'
<b>_ (underscore)</b>	replaces single character	<b>LIKE '_h%'</b> will match Chainring Bolts, Thin-Jam Hex Nut or any other string whose second character is 'b'
<b>[]</b>	replaces any single character within the specified range or set of characters:	<b>LIKE '[CH]%'</b> will match Chainring Bolts, HL Grip Tape or any string starting with an 'C' or an 'H'.
<b>[^]</b>	replaces any single character NOT in the specified range or set of characters	<b>LIKE '[^CH]%'</b> will match Adjustable Race, Bearing Ball or any other string <b>not</b> starting with an 'C' or an 'H'

For example, if you are not sure what the exact product name is but you know it does not start with 'A' and that its third character is 'f' then you can use the following query:

```
SELECT Name, ProductNumber
FROM Production.Product
WHERE Name LIKE '[^A]_f%'
```

Result is:

Name	ProductNumber
Reflector	RF-9198

(1 row(s) affected)

- **CONTAINS** – can be used for creating full-text search queries in order to find the exact or literal matches at the word level. To use **CONTAINS** function you need to have a full text index on that column which is used as an argument. Full text indexes will be discussed later in this book so for now we only concern about **CONTAINS** function.

```
CREATE UNIQUE INDEX ui_ProductID ON Production.Product(ProductID);
CREATE FULLTEXT CATALOG ft_ct AS DEFAULT;
CREATE FULLTEXT INDEX ON Production.Product(Name)
KEY INDEX ui_ProductID;

SELECT Name, Color
FROM Production.Product
WHERE CONTAINS(Name, 'Red');
```



Result is:

Name	Color
Paint - Red	NULL
HL Road Frame - Red, 58	Red
Sport-100 Helmet, Red	Red
...	
Road-250 Red, 48	Red
Road-250 Red, 52	Red
Road-250 Red, 58	Red

(39 row(s) affected)

- FREETEXT – is able to find values that match the meaning and not just the exact words or synonyms of the search condition.

```
SELECT Title
FROM Production.Document
WHERE FREETEXT (Document, 'product code quality' );
```

Result is:

Title
Introduction 1
Crank Arm and Tire Maintenance

(2 row(s) affected)

## Logical Operators

• Logical operators AND, OR, and NOT are required when more than one search criteria is specified

```
SELECT ProductNumber, ListPrice, DaysToManufacture
FROM Production.Product
WHERE ListPrice<2000 AND DaysToManufacture=1
```

ProductNumber	ListPrice	DaysToManufacture
BE-2349	0,00	1
BL-2036	0,00	1
.....		
CS-6583	256,49	1
CS-9183	404,99	1
CH-0234	20,24	1
BB-7421	53,99	1
BB-8107	101,24	1
BB-9108	121,49	1

(154 row(s) affected)

If search criteria requires more than one condition to be specified, then those conditions need to be connected with logical operators AND, OR, and NOT. Expression evaluation with logical operators usually results with a Boolean value TRUE or FALSE.

- AND – results with TRUE only when left and right expressions are TRUE. Next query will return all rows from table Products whose ListPrice is less than 2000 AND takes no longer than 1 day to manufacture them.

```
SELECT ProductNumber, ListPrice, DaysToManufacture
FROM Production.Product
WHERE ListPrice<2000 AND DaysToManufacture=1
```

Result is:

ProductNumber	ListPrice	DaysToManufacture
BE-2349	0,00	1
BL-2036	0,00	1
.....		
CS-6583	256,49	1
CS-9183	404,99	1
CH-0234	20,24	1
BB-7421	53,99	1
BB-8107	101,24	1
BB-9108	121,49	1

(154 row(s) affected)

- OR – results with TRUE when either expression is TRUE. Therefore, execution of previous query with OR logical operator will result with 469 rows (including all rows that can be manufactured in 1 day OR whose ListPrice is less than 2000).

```
SELECT ProductNumber, ListPrice, DaysToManufacture
FROM Production.Product
WHERE ListPrice<2000 OR DaysToManufacture=1
```

Result is:

ProductNumber	ListPrice	DaysToManufacture
AR-5381	0,00	0
BA-8327	0,00	0
BE-2349	0,00	1
....		
BB-7421	53,99	1
BB-8107	101,24	1
BB-9108	121,49	1
BK-R19B-44	539,99	4
BK-R19B-48	539,99	4
BK-R19B-52	539,99	4

(469 row(s) affected)

- NOT – results with reversed value of any other Boolean operator. The following query will return all rows whose ListPrice is NOT less than 2000, or in other words it will return all rows whose ListPrice is greater or equal to 2000.

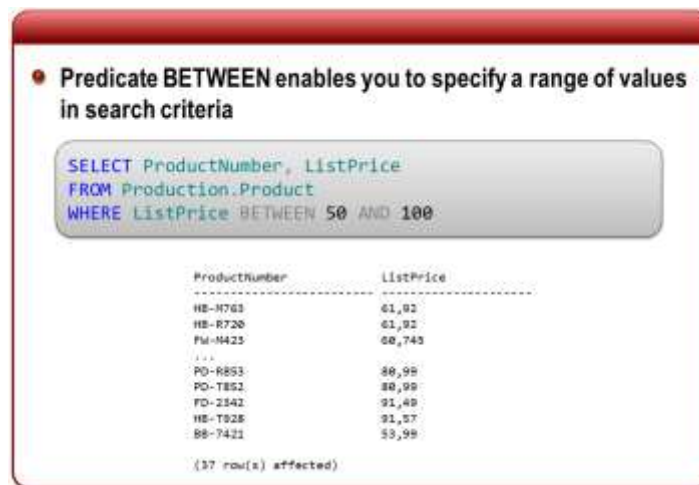
```
SELECT ProductNumber, ListPrice, DaysToManufacture
FROM Production.Product
WHERE NOT ListPrice<2000
```

Result is:

ProductNumber	ListPrice	DaysToManufacture
BK-R93R-62	3578,27	4
BK-R93R-44	3578,27	4
BK-R93R-48	3578,27	4
....		
BK-T79U-50	2384,07	4
BK-T79U-54	2384,07	4
BK-T79U-60	2384,07	4

(35 row(s) affected)

## Retrieving a Range of Values



In order to retrieve the rows whose column value falls within a specified range, you can use operators greater or equal than ( $\geq$ ) and less or equal than ( $\leq$ ). However, range can also be specified by using comparison predicate BETWEEN which returns TRUE if the evaluated expression is greater or equal to the value of the start expression, and is less than or equal to the value of the end expression. The following query retrieves only those rows whose ListPrice is in the range from 50 to 100 (including 50 and 100).

```
SELECT ProductNumber, ListPrice
FROM Production.Product
WHERE ListPrice BETWEEN 50 AND 100
```

Result is:

ProductNumber	ListPrice
HB-M763	61,92
HB-R720	61,92
FW-M423	60,745
...	
PD-M562	80,99
PD-R563	62,09
PD-R853	80,99
PD-T852	80,99
FD-2342	91,49
HB-T928	91,57
BB-7421	53,99

(37 row(s) affected)

The same result set can be retrieved by using the following query search condition:

```
...
WHERE ListPrice >= 50 AND ListPrice <=100
```

## Retrieving a List of Values

• Predicate IN compares a column value to a list of literal values and returns TRUE if compared value matches at least one of the values in the list

```
SELECT ProductNumber, ListPrice
FROM Production.Product
WHERE ListPrice IN (52.64, 74.99, 147.14)
```

ProductNumber	ListPrice
SA-M237	147,14
SA-R430	147,14
SA-T612	147,14
TG-W091-S	74,99
TG-W091-M	74,99
TG-W091-L	74,99
SE-M940	52,64
SE-R995	52,64
SE-T924	52,64

(9 row(s) affected)

In some cases there is necessity that rows in a result set match to the one of values within a particular set and can't be easily defined with a range (defined with a start and end expression). As an efficient solution you can use the IN predicate which compares a column value to a list of literal values and returns TRUE if compared value matches at least one of the values in the list. For example, the following query will retrieve data about all products whose ListPrice is 52.64, 79.99, or 147.14.

```
SELECT ProductNumber, ListPrice
FROM Production.Product
WHERE ListPrice IN (52.64, 74.99, 147.14)
```

Result is:

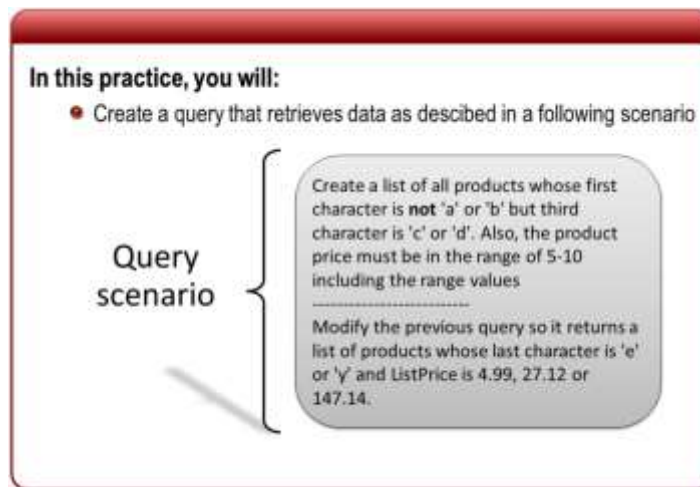
ProductNumber	ListPrice
SA-M237	147,14
SA-R430	147,14
SA-T612	147,14
TG-W091-S	74,99
TG-W091-M	74,99
TG-W091-L	74,99
SE-M940	52,64
SE-R995	52,64
SE-T924	52,64

(9 row(s) affected)

The IN predicate is very similar to the OR operator and therefore same result set can be retrieved by using the following query search condition:

```
...
WHERE ListPrice=52.64 OR ListPrice=74.99 OR ListPrice=147.14
```

## Practice: Filtering Data



**In this practice, you will:**

- Create a query that retrieves data as described in a following scenario

**Query scenario**

Create a list of all products whose first character is **not** 'a' or 'b' but third character is 'c' or 'd'. Also, the product price must be in the range of 5-10 including the range values

Modify the previous query so it returns a list of products whose last character is 'e' or 'y' and ListPrice is 4.99, 27.12 or 147.14.

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 2 from Module 2.

To successfully complete the exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - <http://msftdbprodsamples.codeplex.com/>

### Exercise 2: Filtering Data

7. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
8. On the Connect to Server dialog windows, under Server name type the name of your local instance.
  - a. If it is default then just type (local)
9. Use Windows Authentication
10. Open File menu →New →Database Engine Query

#### *Query Scenario*

Create a list of all products whose first character is **not** 'a' or 'b' but third character is 'c' or 'd'. Also, the product price must be in the range of 5-10 including the range values.

11. Type the following TSQL code in query windows:

```
SELECT Name, ListPrice
FROM Production.Product
WHERE Name LIKE '[^AB]_[CD]%' AND ListPrice BETWEEN 5 AND 10
```

12. Click on Execute or press F5 to run

Result is:

Name	ListPrice
Racing Socks, M	8,99
Racing Socks, L	8,99

(2 row(s) affected)

Modify the previous query so it returns a list of products whose last character is 'e' or 'y' and ListPrice is 4.99, 27.12 or 147.14.

13. Type the following TSQL code in query windows:

```
SELECT Name, ListPrice
FROM Production.Product
WHERE Name LIKE '%[EY]' AND ListPrice IN(4.99, 27.12, 147.14)
```

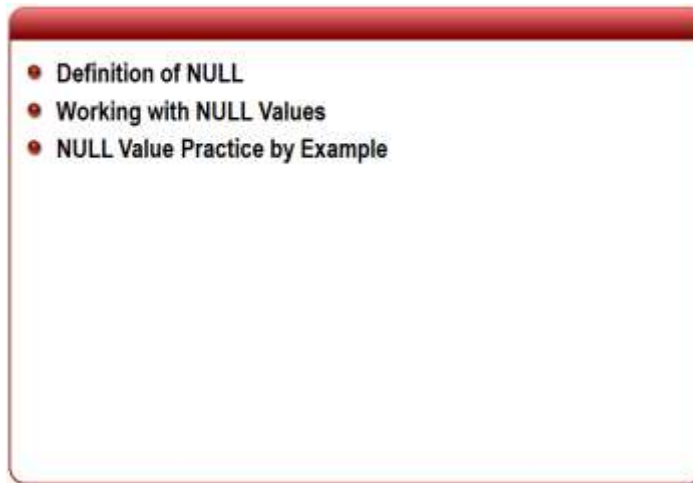
Result is:

Name	ListPrice
ML Mountain Seat Assembly	147,14
ML Road Seat Assembly	147,14
ML Touring Seat Assembly	147,14
LL Mountain Seat/Saddle	27,12
LL Road Seat/Saddle	27,12
LL Touring Seat/Saddle	27,12
Mountain Tire Tube	4,99
Touring Tire Tube	4,99

(8 row(s) affected)

Using wildcard characters at the beginning of the LIKE clause is not considered as a good practice and will be discussed in the last lesson of this module.

## Lesson 3: What is NULL



One of the issues that occur regularly when working with databases is absence, incomplete or unavailable value. In order to overcome these issues SQL uses so called 3-valued logic where expression can either have a value, have no value (NULL) or be UNKNOWN (caused by existence of NULL value in the expression). A NULL is an undefined value and it's usually used as a temporary value that will later be updated with some real data. In context of numeric or string data, a NULL is not the same as zeros or blanks since they are both defined values. In this lesson you will learn how to select and handle NULL values.

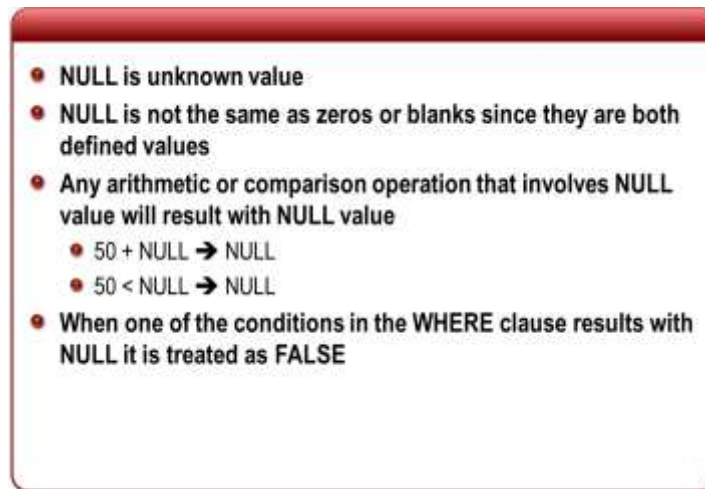
### Objectives

After completing this lesson, you will be able to:

- Filter data with NULL values
- Replace NULL values when printing query results
- Use functions to work with NULL values



## Definition of NULL



Since NULL value differs from any other value, it sometimes requires different forms of handling. Considering the fact that NULL is used to represent missing value, there are several rules that should be borne in mind:

- Any arithmetic or comparison operation that involves NULL value will result with NULL value
  - $50 + \text{NULL} \rightarrow \text{NULL}$
  - $50 < \text{NULL} \rightarrow \text{NULL}$
- When one of the conditions in the WHERE clause results with NULL it is treated as FALSE

## Working with NULL Values

- NULL values can't be compared using standard comparison operators
- Keywords IS NULL and IS NOT NULL are used to identify NULL values

```
SELECT Name, Color
FROM Production.Product
WHERE Color IS NULL
```

Name	Color
Adjustable Race	NULL
Bearing Ball	NULL
BB Ball Bearing	NULL
....	
LL Bottom Bracket	NULL
HL Bottom Bracket	NULL
HL Bottom Bracket	NULL

(248 row(s) affected)

In previous lessons you saw the results of query execution that contained the NULL values. However, in most of the cases you will want to avoid these kinds of results and use mechanisms that will allow you to filter the NULL data. It is important to note that NULL values can't be compared using standard comparison operators and therefore you will have to use keyword IS NULL. The following query will return all rows whose color is NULL.

```
SELECT Name, Color
FROM Production.Product
WHERE Color IS NULL
```

Result is:

Name	Color
Adjustable Race	NULL
Bearing Ball	NULL
BB Ball Bearing	NULL
....	
HL Road Tire	NULL
Touring Tire	NULL
LL Touring Handlebars	NULL
HL Touring Handlebars	NULL
LL Bottom Bracket	NULL
ML Bottom Bracket	NULL
HL Bottom Bracket	NULL

(248 row(s) affected)

The keyword IS NULL can also be used in combination with negation NOT, so you can write this condition:

```
WHERE Color IS NOT NULL
```

## NULL Value Practice by Example

- ISNULL** – if the argument value is NULL it converts it to some other meaningful value defined as the second argument.
 

```
SELECT Name, ISNULL(Color, 'N/A')
FROM Production.Product
```
- NULLIF** – returns NULL if two compared arguments match; otherwise the first argument is returned.
 

```
SELECT JobTitle, NULLIF(SickLeaveHours, VacationHours)
FROM HumanResources.Employee
```
- COALESCE** – accepts a set of parameters and return the first parameter that is not NULL.
 

```
SELECT ProductNumber, ProductLine, Class, Style,
COALESCE(ProductLine, Class, Style)
FROM Production.Product
```

In the previous part of the lesson we used the IS NULL and IS NOT NULL keywords to filter information that is currently unavailable. However, sometimes you want to make NULL values part of the result set but in a form that is understandable to people who are not engaged in the field of information technology, or rather the databases (managers, economists). One way is to adjust the NULL value to the end user and replace it with other more appropriate terms such as N/A (Not Applicable). These adjustments can be made by using ISNULL, NULLIF or COALESCE functions.

- ISNULL** – if the argument value is NULL, it converts it to some other meaningful value defined as the second argument. It is important to emphasize that ISNULL function is not a standard and it is recommended to use COALESCE instead.

```
SELECT Name, ISNULL(Color, 'N/A')
FROM Production.Product
```

Result is:

Name	
Adjustable Race	N/A
Bearing Ball	N/A
BB Ball Bearing	N/A
....	
Mountain-500 Black, 52	Black
LL Bottom Bracket	N/A
ML Bottom Bracket	N/A
HL Bottom Bracket	N/A
Road-750 Black, 44	Black
Road-750 Black, 48	Black
Road-750 Black, 52	Black

(504 row(s) affected)

- NULLIF** – returns NULL if two compared arguments match; otherwise the first argument is returned.

```

SELECT JobTitle,
       NULLIF(SickLeaveHours, VacationHours)
FROM HumanResources.Employee

```

Result is:

JobTitle	
Chief Executive Officer	69
...	
Production Technician - WC60	NULL
...	
Sales Representative	NULL
Sales Representative	38

(290 row(s) affected)

- COALESCE – accepts a set of parameters and returns the first parameter that is not NULL. COALESCE function returns a NULL if a set contains only NULL values.

```

COALESCE (NULL, NULL, NULL)      --> Returns NULL
COALESCE ( 'Argument1', NULL, NULL) --> Returns Argument1
COALESCE (NULL, 'Argument2', NULL) --> Returns Argument2
COALESCE (NULL, NULL, 'Argument3') --> Returns Argument3

```

The last column of the following query will display first not NULL value found in columns ProductLine, Class, or Style.

```

SELECT ProductNumber, ProductLine, Class, Style,
       COALESCE(ProductLine, Class, Style)
FROM Production.Product

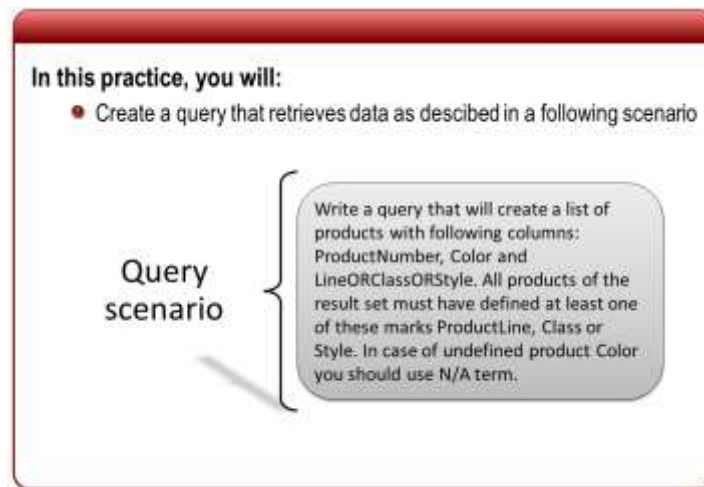
```

Result is:

ProductNumber	ProductLine	Class	Style
AR-5381	NULL	NULL	NULL
CA-5965	NULL	L	NULL
CA-6738	NULL	M	NULL
CA-7457	NULL	NULL	NULL
.....			
BK-M18B-48	M	L	U
BK-M18B-52	M	L	U
BB-7421	NULL	L	NULL
BB-8107	NULL	M	NULL
BB-9108	NULL	H	NULL
BK-R19B-44	R	L	U

(504 row(s) affected)

## Practice: Working with NULL



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 3 from Module 2.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - <http://msftdbprodsamples.codeplex.com/>

### Exercise 3: Working with NULL

14. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
15. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
  - a. If it is default then just type (local)
16. Use Windows Authentication
17. Open File menu →New →Database Engine Query

#### *Query Scenario*

Write a query that will create a list of products with the following columns: ProductNumber, Color and LineORClassORStyle. All products of the result set must have defined at least one of these marks ProductLine, Class or Style. In case of undefined product Color you should use N/A term.

18. Type the following TSQL code in query windows:

```
SELECT ProductNumber, ISNULL(Color, 'N/A') as Color,  
COALESCE(ProductLine, Class, Style) AS LineORClassORStyle  
FROM Production.Product  
WHERE COALESCE(ProductLine, Class, Style) IS NOT NULL
```

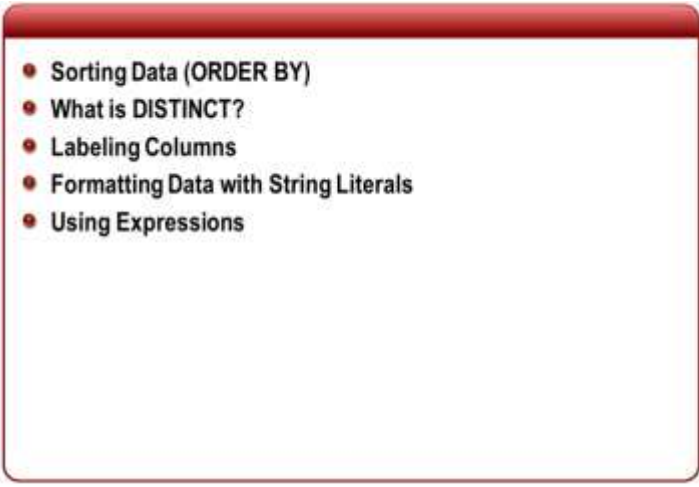
19. Click on Execute or press F5 to run

Result is:

ProductNumber	Color	LineORClassORStyle
CA-5965	Black	L
CA-6738	Black	M
GT-0820	N/A	L
...		
BK-R19B-44	Black	R
BK-R19B-48	Black	R
BK-R19B-52	Black	R

(308 row(s) affected)

## Lesson 4: Formatting Output

- 
- Sorting Data (ORDER BY)
  - What is DISTINCT?
  - Labeling Columns
  - Formatting Data with String Literals
  - Using Expressions

With current knowledge of SQL you are certainly able to select the required data stored in a database table. However, if we are talking about larger production databases, at first glance the selected data would probably not be fully understandable and usable. To overcome this issue you need to do additional formatting such as sorting, eliminating duplicate rows, creating custom column labels, etc. By using the aforementioned formatting, you are able to arrange data into appropriate form.

### Objectives

After completing this lesson, you will be able to:

- Sort data by using the ORDER BY clause
- Eliminate duplicated rows by using DISTINCT keyword
- Label columns in result sets
- Use string literals
- Use expressions in SELECT and WHERE clauses

## Sorting Data (ORDER BY)

- **ORDER BY clause is used to sort the records in a result set**
- **Two forms of sorting**
  - ascending (keyword ASC) – default
  - descending (keyword DESC)

```
SELECT TOP 100 FirstName, LastName
FROM Person.Person
ORDER BY FirstName
```

- **Sorting can also be carried out on multiple columns**

```
SELECT TOP 100 FirstName, LastName
FROM Person.Person
ORDER BY FirstName ASC, LastName DESC
```

By default, records in a result set are ordered in a way in which they were entered into the table. If you want to modify the default order of records in a result set, you can use the ORDER BY clause. This clause enables you to specify ascending (keyword ASC) or descending (keyword DESC) form of ordering and must be defined as the last clause of the SELECT statement. In the next example we will create a list of employees ordered by their first name. Since the following query returns a large number of records (19972) we will use keyword TOP 100 in the SELECT clause to narrow the result set.

```
SELECT TOP 100 FirstName, LastName
FROM Person.Person
ORDER BY FirstName
```

Result is:

FirstName	LastName
A.	Leonetti
A.	Wright
A. Scott	Wright
Aaron	Adams
Aaron	Alexander
....	
Abigail	Diaz
Abigail	Flores
Abigail	Foster

(100 row(s) affected)

As you can see the default form of order is ascending (ASC) and if you want the descending form you will need to specify the keyword DESC. In addition to simple sorting by one column, sorting can be carried out on multiple columns where each can have a different form of sorting.



```
SELECT TOP 100 FirstName, LastName
FROM Person.Person
ORDER BY FirstName ASC, LastName DESC
```

Result is:

FirstName	LastName
A.	Wright
A.	Leonetti
A. Scott	Wright
Aaron	Zhang
Aaron	Young
....	
Abigail	Rogers
Abigail	Rodriguez
Abigail	Robinson

(100 row(s) affected)

It is important to note that column names used in ORDER BY clause do not have to be listed inside of the SELECT clause.

## What is DISTINCT?



In order to make sure there are no duplicated records in your result set, you can use DISTINCT keyword. The DISTINCT keyword is always placed immediately after the SELECT clause. In the next example we want to display only unique product colors.

```

SELECT DISTINCT Color
FROM Production.Product

```

Result is:

```

Color
-----
NULL
Black
Blue
Grey
Multi
Red
Silver
Silver/Black
White
Yellow

(10 row(s) affected)

```

As you can see, the result set contains one instance of each unique row. However, it is important to note that DISTINCT only ensures uniqueness of the column values listed in the SELECT clause and ignores other columns that may also have unique values.

If you want to add one or more columns in the result set then you have to bear in mind that the DISTINCT keyword is considering uniqueness at the row level, including values of all columns listed in the SELECT clause.

```

SELECT DISTINCT Color, Name
FROM Production.Product

```

Result is:

Color	Name
-----	-----
NULL	Adjustable Race
NULL	Bearing Ball
...	
Black	Road-750 Black, 48
Black	Road-750 Black, 52
(504 row(s) affected)	

## Labeling Columns

- Keyword AS enables you to specify the alias which is a short substitute or nickname for a column or a table name
- Column labels can be specified by using quotes, brackets or equal sign

```
SELECT Name AS "Product Name",
       ProductNumber AS [Product Number],
       [Size Unit Code] = SizeUnitMeasureCode
FROM Production.Product
```

- Table aliases are very useful when querying multiple tables

```
SELECT P.Name, P.ProductNumber AS Number
FROM Production.Product AS P
```

Those engaged in the database development sometimes use obscure and long column names. In order to change these names when printing the query results, you need to use the keyword AS which specifies the alias. Alias is a short substitute or nickname for a column or a table name.

```
SELECT Name, ProductNumber AS Number
FROM Production.Product
```

Result is:

Name	Number
Adjustable Race	AR-5381
Bearing Ball	BA-8327
BB Ball Bearing	BE-2349
Headset Ball Bearings	BE-2908
Blade	BL-2036
....	
ML Bottom Bracket	BB-8107
HL Bottom Bracket	BB-9108
Road-750 Black, 44	BK-R19B-44
Road-750 Black, 48	BK-R19B-48
Road-750 Black, 52	BK-R19B-52

(504 row(s) affected)

Alias which is composed of several words must be put inside of double quotes "Product Number" or brackets [Product Number].

```
SELECT Name AS "Product Name",
       ProductNumber AS [Product Number]
FROM Production.Product
```

Alias can also be created by omitting the AS keyword or by using equal sign.

```
SELECT Name "Product Name",  
       [Product Number] = ProductNumber  
FROM Production.Product
```

Using aliases for the long table names is very useful when you create a query to retrieve data from multiple tables which will be discussed in the module 4. The following example creates alias P for table Product and uses it in SELECT clause to reference column names.

```
SELECT P.Name, P.ProductNumber AS Number  
FROM Production.Product AS P
```

Result is:

Name	Number
Adjustable Race	AR-5381
Bearing Ball	BA-8327
....	
Road-750 Black, 44	BK-R19B-44
Road-750 Black, 48	BK-R19B-48
Road-750 Black, 52	BK-R19B-52

(504 row(s) affected)

## Formatting Data with String Literals

• Literals are used to format query results in order to increase data readability

```
SELECT 'Product ', Name, ' has a list price of ', ListPrice
FROM Production.Product
```

	Name	ListPrice
Product	Adjustable Race	has a list price of 0,00
Product	Bearing Ball	has a list price of 0,00
Product	BB Ball Bearing	has a list price of 0,00
....		
Product	Road-750 Black, 44	has a list price of 539,99
Product	Road-750 Black, 48	has a list price of 539,99
Product	Road-750 Black, 52	has a list price of 539,99

(584 row(s) affected)

In almost the same manner as aliases allows us to create more descriptive column names, literals are used to format query results in order to increase data readability. Literals are constant values that can be characters, numbers and other symbols. The most basic form of using literals is shown in the next example.

```
SELECT 'Product ', Name, ' has a list price of ', ListPrice
FROM Production.Product
```

Result is:

	Name	ListPrice
Product	Adjustable Race	has a list price of 0,00
Product	Bearing Ball	has a list price of 0,00
Product	BB Ball Bearing	has a list price of 0,00
....		
Product	Road-750 Black, 44	has a list price of 539,99
Product	Road-750 Black, 48	has a list price of 539,99
Product	Road-750 Black, 52	has a list price of 539,99

(584 row(s) affected)

Literals can also be combined with other functions in order to create single column values.

## Using Expressions

• Expressions can be used in SELECT and WHERE clauses to create necessary data adjustment

```
SELECT ('Product ' + ProductNumber) AS Number,
       ListPrice AS [Old price],
       (ListPrice * 1.17) AS [New price]
FROM Production.Product
WHERE ListPrice > 0
```

Number	Old price	New price
Product SA-M198	133,34	156.007800
Product SA-M237	147,14	172.153800
Product SA-M687	196,92	230.396400
...		
Product BK-R19B-44	539,99	631.788300
Product BK-R19B-48	539,99	631.788300
Product BK-R19B-52	539,99	631.788300

(304 row(s) affected)

While formatting data output, in many cases you will be required to perform some sort of mathematical operators such as addition, multiplication, subtraction, etc. Mathematical expressions can be used in SELECT and WHERE clauses which are very useful for creating more descriptive results and eliminate necessity for data adjustment after query results have been delivered.

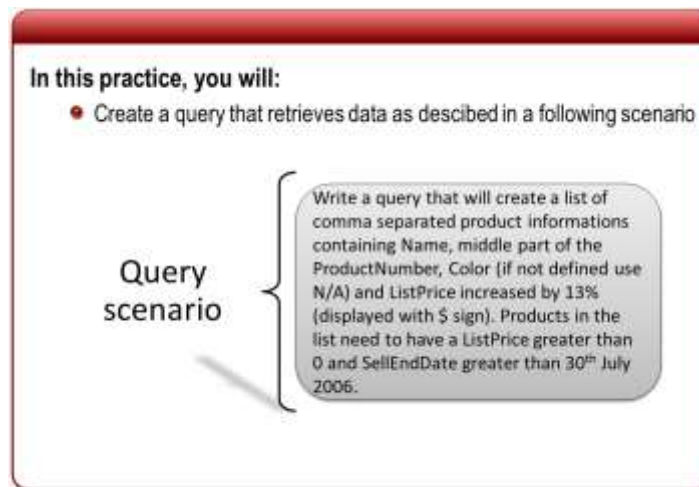
```
SELECT ('Product ' + ProductNumber) AS Number,
       ListPrice AS [Old price],
       (ListPrice * 1.17) AS [New price]
FROM Production.Product
WHERE ListPrice > 0
```

Result is:

Number	Old price	New price
Product SA-M198	133,34	156.007800
Product SA-M237	147,14	172.153800
Product SA-M687	196,92	230.396400
...		
Product BK-R19B-44	539,99	631.788300
Product BK-R19B-48	539,99	631.788300
Product BK-R19B-52	539,99	631.788300

(304 row(s) affected)

## Practice: Formatting the Query Output



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 3 from Module 2.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - <http://msftdbprodsamples.codeplex.com/>

### Exercise 4: Formatting the Query Output

1. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
2. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
  - a. If it is default then just type (local)
3. Use Windows Authentication
4. Open File menu →New →Database Engine Query

#### *Query Scenario*

Write a query that will create a list of comma separated product informations containing Name, middle part of the ProductNumber, Color (if not defined use N/A) and ListPrice increased by 13% (displayed with \$ sign). Products in the list need to have a ListPrice greater than 0 and SellEndDate greater than 30<sup>th</sup> July 2006.



5. Type the following TSQL code in query windows:

```
SELECT Concat (Name, ', ',  
              SUBSTRING (ProductNumber, 4, LEN (ProductNumber) - 3) ,  
              ', ', ISNULL (Color, 'N/A') , ', ', ListPrice, '$')  
      AS [Product list]  
FROM Production.Product  
WHERE ListPrice > 0 AND SellEndDate > '2006-06-30'
```

6. Click on Execute or press F5 to run

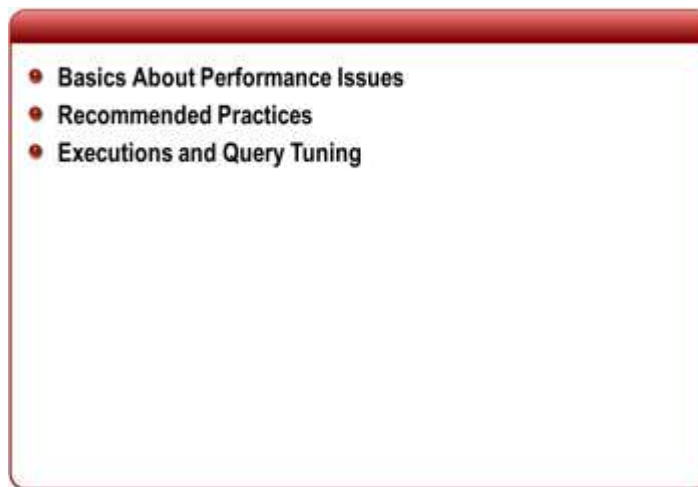
Result is:

Product list

```
-----  
LL Road Frame - Red, 44,R38R-44,Red,337.22$  
LL Road Frame - Red, 48,R38R-48,Red,337.22$  
....  
Men's Bib-Shorts, S,M891-S,Multi,89.99$  
Men's Bib-Shorts, M,M891-M,Multi,89.99$  
Men's Bib-Shorts, L,M891-L,Multi,89.99$  
Full-Finger Gloves, S,F110-S,Black,37.99$  
Full-Finger Gloves, M,F110-M,Black,37.99$  
Full-Finger Gloves, L,F110-L,Black,37.99$
```

(69 row(s) affected)

## Lesson 5: Performance and Best Practices



Depending on the database environment, the speed and efficiency of query execution is sometimes of great importance. This is particularly true for databases that are used to support applications intended for permanent data manipulation such as bank systems, plane tickets reservation and etc. Query execution mostly depends on the query designer and therefore it is very important to understand some of the most common elements that slow down the queries execution. However, one of the mitigating circumstances is the fact that the database management systems such as MS SQL 2012 provides a number of tools that helps you identify the weak queries points.

### Objectives

After completing this lesson, you will be able to:

- Understand Basics About Performance Issues
- Apply Recommended Practices
- Perform Query Tuning

## Basics About Performance Issues

- Poorly written query → bad performance → negative user experience
- Remember: database grows over time
- Follow a very simple set of rules → better performing queries
- Monitor performance
  - Starting point: diagnose performance issues with specific queries

```
SELECT * FROM sys.dm_os_wait_stats
WHERE wait_time_ms > 0
```

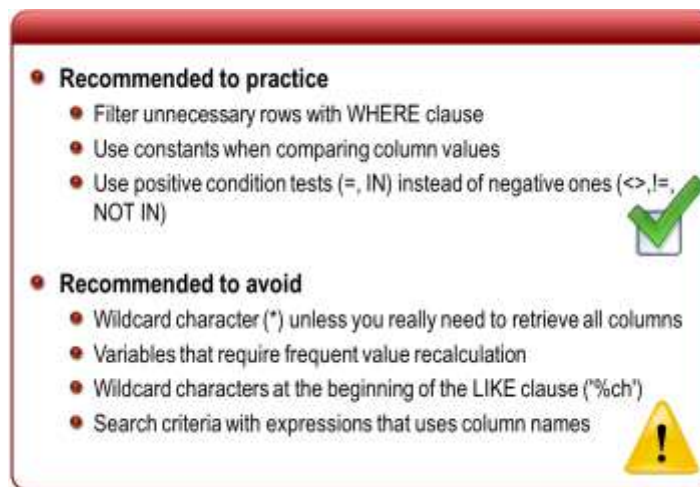
A poorly written query leads to bad performance and thus contributes to creating of a negative user experience. One important note you should have in mind when creating a queries is that database grows over time and that the initial factors that served you as guidance are not always valid. For example, queries that are executed very fast at the start do not guarantee that after a certain period of time will not cause degradation of system performance.

When writing a query you should follow a very simple set of rules which are presented later in this lesson but before that we will mention the notion of the index and how they can contribute to better performing queries. Indexes are much the same as book indexes, providing the database with quick jump points on where to find the full reference to searched data. Therefore if it is possible you should use indexed columns when creating search conditions in the WHERE clause. At certain time intervals, it is good practice to rebuild the index table in order to update the references.

The way you can certainly keep track the query execution is by monitoring performance. As a starting point in monitoring performance you can use following query that returns information about all the waits encountered by threads that executed and diagnose performance issues with specific queries.

```
SELECT * FROM sys.dm_os_wait_stats
WHERE wait_time_ms > 0
```

## Recommended Practices



Considering the processing power of modern computer systems significant slowdown can primarily be experienced in cases when working with complex queries and larger amounts of data. Practice has shown that by following the simple set of rules you can substantially avoid slow and inefficient queries.

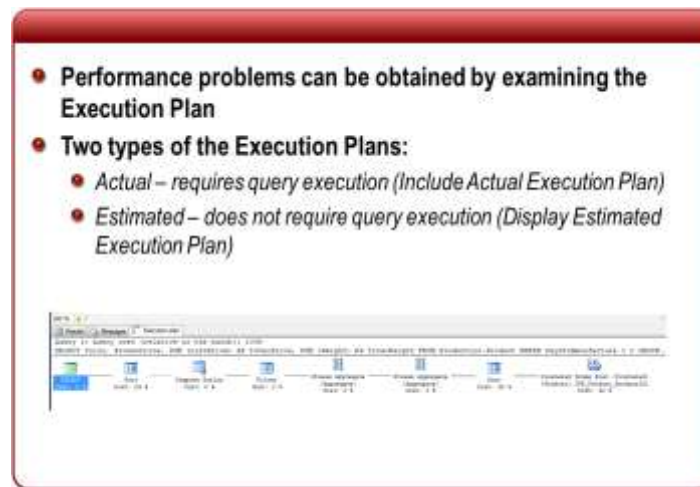
Recommended to practice:

- Filter unnecessary rows with WHERE clause
- Use constants when comparing column values
- Use positive condition tests (=, IN) instead of negative ones (<>, !=, NOT IN)

Recommended to avoid:

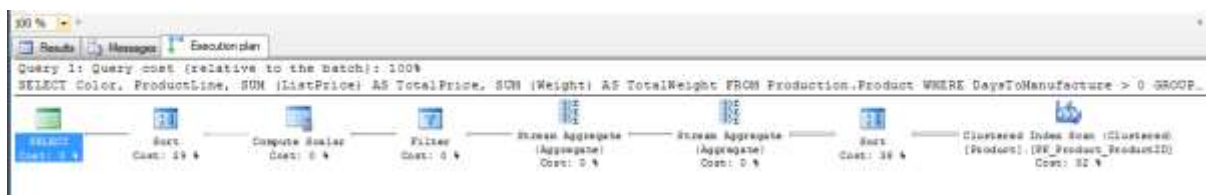
- Wildcard character (\*) unless you really need to retrieve all columns
- Variables that require frequent value recalculation
- Wildcard characters at the beginning of the LIKE clause ('%ch')
- Search criteria with expressions that uses column names

## Executions and Query Tuning



One of the tools that will surely become your good friend is called a Microsoft SQL Server Profiler. This tool is a graphical user interface which enables you to monitor an instance of the Database Engine in order to capture and save data about each event. For example, you can monitor a production environment to see which queries or stored procedures are affecting performance by executing too slowly. Information gathered by Profiler can be analysed with tool called Database Engine Tuning Advisor and will be discussed in the Practice part of the lesson.

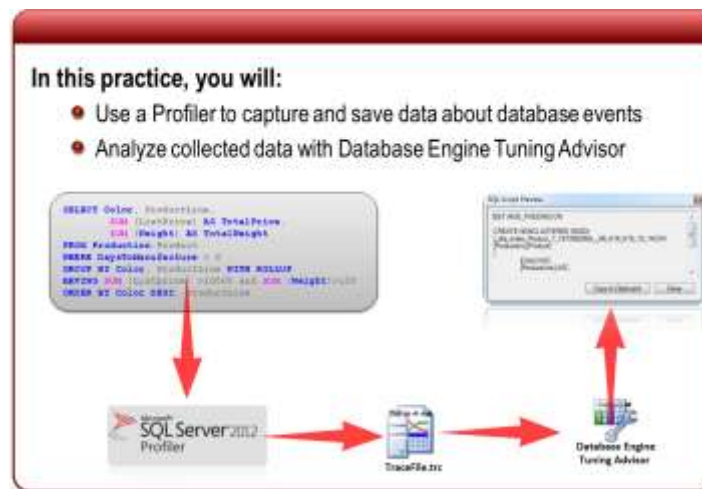
Besides Profiler, cause of performance problems can also be obtained by examining the Execution Plan which provides you with detailed information about query execution. In order to get insight into the execution plan is necessary to activate option *Include Actual Execution Plan* (shortcut Ctrl + M or Query -> Include Actual Execution Plan). After that, besides Results and Messages tab, query execution will provide you with a new tab called Execution plan. This tab contains a graphical representation of individual steps in the query execution.



To get more familiar with this functionality try to follow the execution plan for queries that combine different forms of search conditions, sorting and etc. It is important to try to recognize the difference of using indexed and non-indexed columns in search criteria. From the previous figure we see that data sorting required 67% of the entire query execution process.

*Actual Execution Plan* generation requires query to be executed. If you are testing your query at the time of database intensive use you may consider using *Estimated Execution Plan* (shortcut Ctrl + L or Query -> Display Estimated Execution Plan) that does not require the execution of queries.

## Practice: Query Tuning



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 5 from Module 2.

To successfully complete exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
  - <http://msftdbprodsamples.codeplex.com/>

### Exercise 5: Query Tuning

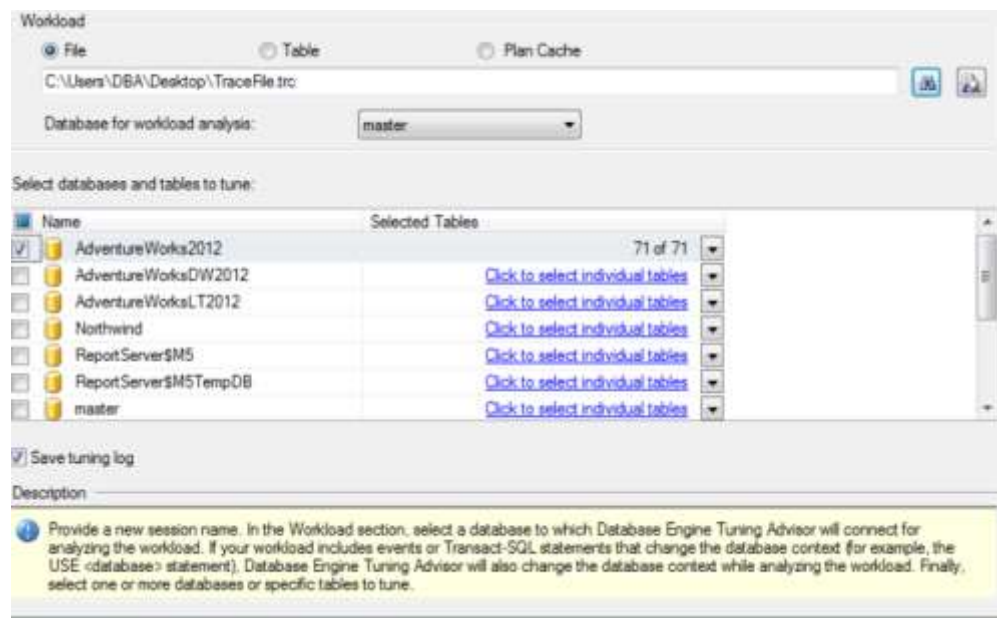
Below we show how to use Profiler, and how you can use the information it generates.

- Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
- On the Connect to Server dialog windows, under Server name type the name of your local instance.
  - If it is default then just type (local)
- Use Windows Authentication
- Click Tools->SQL Server Profiler
  - On the Connect to Server dialog windows, under Server name type the name of your local instance.
    - If it is default then just type (local)
  - Use Windows Authentication
- On the Trace Properties dialog window, under *Use the template* combo box select *Tuning*
- Check *Save to a file* option and specify file name (for example: *TraceFile.trc*) and location

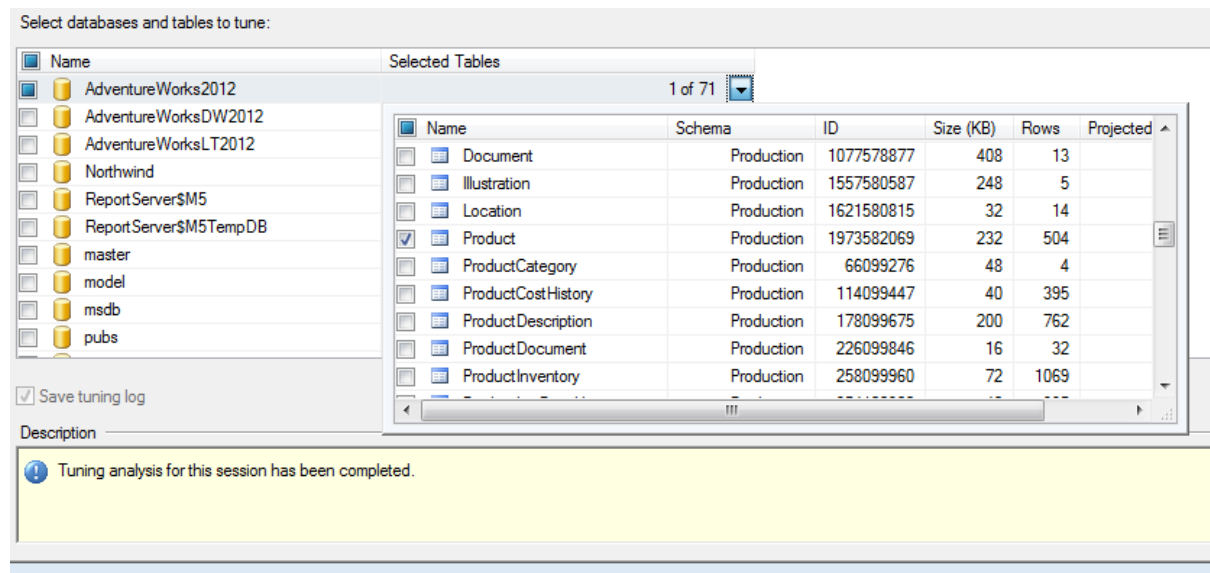
13. Go back to SQL Server Management Studio and execute following query. Parts of the query may not be fully understandable at this moment but they will be after you study next few modules.

```
SELECT Color, ProductLine,
       SUM (ListPrice) AS TotalPrice,
       SUM (Weight) AS TotalWeight
FROM Production.Product
WHERE DaysToManufacture > 0
GROUP BY Color, ProductLine WITH ROLLUP
HAVING SUM (Listprice) >10000 and SUM (Weight)>100
ORDER BY Color DESC, ProductLine
```

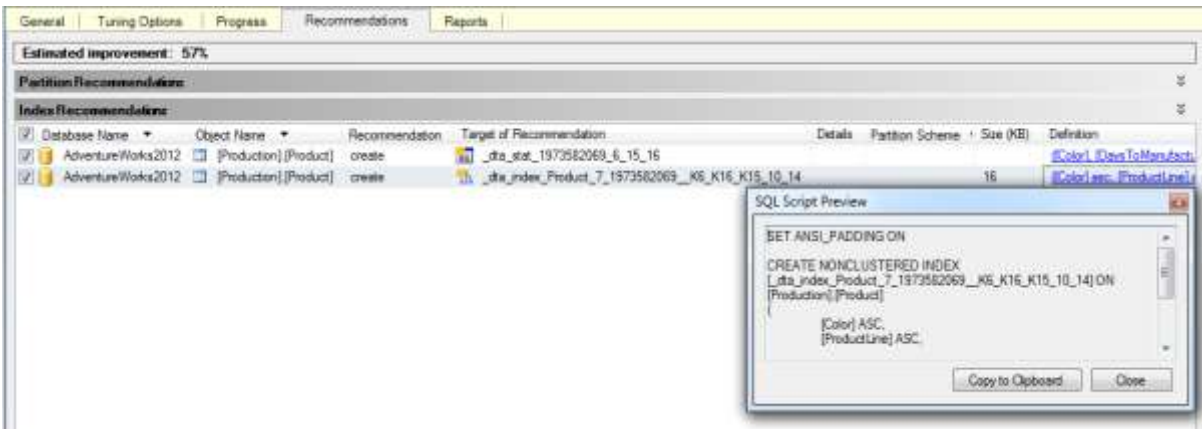
14. SQL Server Profiler will record every execution part of the previous query so you can use it for further analysis.
15. Open Database Engine Tuning Advisor (Click Start->Microsoft SQL Server 2012->Performance Tools->Database Engine Tuning Advisor)
16. Load the Workload file you created with SQL Server Profiler (*TraceFile.trc*)
17. Select the database AdventureWorks 2012 and from the objects list select a Production table



18. Click the Start Analysis
19. After analysis is completed, click on the Recommendation

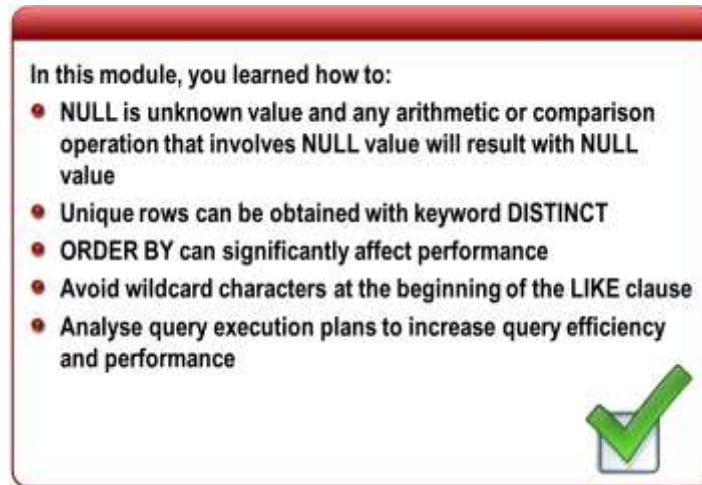


- 20. Analyse the results
- 21. According to the executed query Tuning Advisor created two recommendations which, if are implemented, could raise query performance by 57%





## Summary



In this module we looked at the `SELECT` statement as the most fundamental one which provides the ability to retrieve data from a database. Data retrieval requires specification of data that you want retrieve (column names) and data source by using the `FROM` clause. In order to limit the number of rows returned in the result set you can define a necessary conditions in the `WHERE` clause. Greater number of conditions needs to be connected by using logical operators `AND`, `OR`, and `NOT`, which usually results with either true or false Boolean value. Particular order of data in a result set can be achieved by using the `ORDER BY` keyword.

In addition to the aforementioned key points you should remember from this module are:

- Avoid `SELECT *` unless you really need to retrieve all columns
- `NULL` is unknown value and any arithmetic or comparison operation that involves `NULL` value will result with `NULL` value
- Unique rows can be obtained with keyword `DISTINCT`
- Sorting can significantly affect performance
- Avoid wildcard characters at the beginning of the `LIKE` clause
- Analyse query execution plans to increase query efficiency and performance