# Database Systems(2)

# Tutorial 5

**DR. ALYAA HAMZA**

**ENG. ESRAA SHEHAB**

# SQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records.

## EXISTS Syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

# The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

# HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

# Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

# More HAVING Examples

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Employees" table:

| EmployeeID | LastName | FirstName | BirthDate | Photo | Notes |
|---|---|---|---|---|---|
| 1 | Davolio | Nancy | 1968-12-08 | EmpID1.pic | Education includes a BA.... |
| 2 | Fuller | Andrew | 1952-02-19 | EmpID2.pic | Andrew received his BTS.... |
| 3 | Leverling | Janet | 1963-08-30 | EmpID3.pic | Janet has a BS degree.... |

```sql
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM  (Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

Number of Records: 8

| LastName | NumberOfOrders |
|---|---|
| Buchanan | 11 |
| Callahan | 27 |
| Davolio | 29 |
| Fuller | 20 |
| King | 14 |
| Leverling | 31 |
| Peacock | 40 |
| Suyama | 18 |

# Difference between HAVING and WHERE Clause

| HAVING | WHERE |
|---|---|
| 1. The HAVING clause is used in database systems to fetch the data/values from the groups according to the given condition. | 1. The WHERE clause is used in database systems to fetch the data/values from the tables according to the given condition. |
| 2. The HAVING clause is always executed with the GROUP BY clause. | 2. The WHERE clause can be executed without the GROUP BY clause. |
| 3. The HAVING clause can include SQL aggregate functions in a query or statement. | 3. We cannot use the SQL aggregate function with WHERE clause in statements. |
| 4. We can only use SELECT statement with HAVING clause for filtering the records. | 4. Whereas, we can easily use WHERE clause with UPDATE, DELETE, and SELECT statements. |
| 5. The HAVING clause is used in SQL queries after the GROUP BY clause. | 5. The WHERE clause is always used before the GROUP BY clause in SQL queries. |
| 6. We can implements this SQL clause in column operations. | 6. We can implements this SQL clause in row operations. |
| 7. It is a post-filter. | 7. It is a pre-filter. |
| 8. It is used to filter groups. | 8. It is used to filter the single record of the table. |

**Example 1:** Let's take the following **Employee** table, which helps you to analyze the **HAVING** clause with **SUM** aggregate function:

| Emp_Id | Emp_Name | Emp_Salary | Emp_City |
| --- | --- | --- | --- |
| 201 | Abhay | 2000 | Goa |
| 202 | Ankit | 4000 | Delhi |
| 203 | Bheem | 8000 | Jaipur |
| 204 Ram | 2000 | Goa | |
| 205 | Sumit | 5000 | Delhi |

If you want to add the salary of employees for each city, you must write the following query:

```
SELECT SUM(Emp_Salary), Emp_City FROM Employee GROUP BY Emp_City;
```

The output of the above query shows the following output:

| SUM(Emp_Salary) | Emp_City |
| --- | --- |
| 4000 | Goa |
| 9000 | Delhi |
| 8000 | Jaipur |

Now, suppose that you want to show those cities whose total salary of employees is more than 5000.
For this case, you must type the following query with the HAVING clause in SQL:

```
SELECT SUM(Emp_Salary), Emp_City
FROM Employee
GROUP BY Emp_City
HAVING SUM(Emp_Salary)>5000;
```

The output of the above SQL query shows the following table in the output:

| SUM(Emp_Salary) | Emp_City |
|---|---|
| 9000 | Delhi |
| 8000 | Jaipur |

**Example 2:** Let's take the following **Employee** table, which helps you to analyze the HAVING clause with MIN and MAX aggregate function:

| Emp_ID | Name | Emp_Salary | Emp_Dept |
|---|---|---|---|
| 1001 | Anuj | 9000 | Finance |
| 1002 | Saket | 4000 | HR |
| 1003 | Raman | 3000 | Coding |
| 1004 | Renu | 6000 | Coding |
| 1005 | Seenu | 5000 | HR |
| 1006 | Mohan | 10000 | Marketing |
| 1007 | Anaya | 4000 | Coding |
| 1008 | Parul | 8000 | Finance |

## MIN Function with HAVING Clause:

If you want to show each department and the minimum salary in each department, you must write the following query:

**SELECT MIN**(Emp_Salary), Emp_Dept **FROM** Employee **GROUP BY** Emp_Dept;

The output of the above query shows the following output:

| MIN(Emp_Salary) | Emp_Dept |
|---|---|
| 8000 | Finance |
| 4000 | HR |
| 3000 | Coding |
| 10000 | Marketing |

Now, suppose that you want to show only those departments whose minimum salary of employees is greater than 4000.

For this case, you must type the following query with the HAVING clause in SQL:

```
SELECT MIN(Emp_Salary), Emp_Dept
FROM Employee
GROUP BY Emp_Dept
HAVING MIN(Emp_Salary) > 4000 ;
```

| MIN(Emp_Salary) | Emp_Dept |
|---|---|
| 8000 | Finance |
| 10000 | Marketing |

In the above employee table, if you want to list each department and the maximum salary in each department.
For this, you must write the following query:

```
SELECT MAX(Emp_Salary), Emp_Dept FROM Employee GROUP BY Emp_Dept;
```

The above query will show the following output:

| MAX(Emp_Salary) | Emp_Dept |
| --- | --- |
| 9000 | Finance |
| 5000 | HR |
| 6000 | Coding |
| 10000 | Marketing |

Now, suppose that you want to show only those departments whose maximum salary of employees is less than 8000.

For this case, you must type the following query with the HAVING clause in SQL:

```
SELECT MAX(Emp_Salary), Emp_Dept
FROM Employee
GROUP BY Emp_Dept
HAVING MAX(Emp_Salary) < 8000 ;
```

The output of the above SQL query shows the following table in the output:

| MAX(Emp_Salary) | Emp_Dept |
| --- | --- |
| 5000 | HR |
| 6000 | Coding |

# IN vs. EXISTS

| SN | IN Operator | EXISTS Operator |
|---|---|---|
| 1. | It is used to minimize the multiple OR conditions. | It is used to check the existence of data in a subquery. In other words, it determines whether the value will be returned or not. |
| 2. | It compares the values between subquery (child query) and parent query. | It does not compare the values between subquery and parent query. |
| 3. | It scans all values inside the IN block. | It stops for further execution once the single positive condition is met. |
| 4. | It can return TRUE, FALSE, or NULL. Hence, we can use it to compare NULL values. | It returns either TRUE or FALSE. Hence, we cannot use it to compare NULL values. |
| 5. | We can use it on subqueries as well as with values. | We can use it only on subqueries. |
| 6. | It executes faster when the subquery result is less. | It executes faster when the subquery result is large. It is more efficient than IN because it processes Boolean values rather than values itself. |
| 7. | `Syntax to use IN clause:`<br>`SELECT col_names`<br>`FROM tab_name`<br>`WHERE col_name IN (subquery);` | `Syntax to use EXISTS clause:`<br>`SELECT col_names`<br>`FROM tab_name`<br>`WHERE [NOT] EXISTS (subquery);` |

# GROUP BY vs. ORDER BY

| SN | GROUP BY | ORDER BY |
|---|---|---|
| 1. | It is used to group the rows that have the same values. | It sorts the result set either in ascending or descending order. |
| 2. | It may be allowed in CREATE VIEW statement. | It is not allowed in CREATE VIEW statement |
| 3. | It controls the presentation of rows. | It controls the presentation of columns. |
| 4. | It is mandatory to use aggregate functions in the GROUP BY. | It's not mandatory to use aggregate functions in the ORDER BY. |
| 5. | It is always used before the ORDER BY clause in the SELECT statement. | It is always used after the GROUP BY clause in the SELECT statement. |
| 6. | Here, the grouping is done based on the similarity among the row's attribute values. | Here, the result-set is sorted based on the column's attribute values, either ascending or descending order. |

## Subquery with SELECT statement

In SQL, inner queries or nested queries are used most frequently with the SELECT statement.

The syntax of Subquery with the SELECT statement is described in the following block:

```
SELECT Column_Name1, Column_Name2, ...., Column_NameN
FROM Table_Name WHERE Column_Name Comparison_Operator
( SELECT Column_Name1, Column_Name2, ...., Column_NameN
FROM Table_Name WHERE condition;
```

**Examples of Subquery with the SELECT Statement**

**Example 1:** This example uses the Greater than comparison operator with the Subquery.

Let's take the following table named Student_Details, which contains Student_RollNo., Stu_Name, Stu_Marks, and Stu_City column.

| Student_RollNo. | Stu_Name | Stu_Marks | Stu_City |
|---|---|---|---|
| 1001 | Akhil | 85 | Agra |
| 1002 | Balram | 78 | Delhi |
| 1003 | Bheem | 87 | Gurgaon |
| 1004 | Chetan | 95 | Noida |
| 1005 | Diksha | 99 | Agra |
| 1006 | Raman | 90 | Ghaziabad |
| 1007 | Sheetal | 68 | Delhi |

The following SQL query returns the record of those students whose marks are greater than the average of total marks:

```sql
SELECT *
FROM Student_Details
WHERE Stu_Marks > ( SELECT AVG(Stu_Marks ) FROM Student_Details);
```

**Output:**

| Student_RollNo. | Stu_Name | Stu_Marks | Stu_City |
|---|---|---|---|
| 1003 | Bheem | 87 | Gurgaon |
| 1004 | Chetan | 95 | Noida |
| 1005 | Diksha | 99 | Agra |
| 1006 | Raman | 90 | Ghaziabad |

## Subquery with the INSERT statement

We can also use the subqueries and nested queries with the INSERT statement in Structured Query Language. We can insert the results of the subquery into the table of the outer query. The syntax of Subquery with the INSERT statement is described in the following block:

```
INSERT INTO Table_Name SELECT * FROM Table_Name WHERE Column_Name Operator (Subquery);
```

### Examples of Subquery with the INSERT Statement

**Example1:**

This example inserts the record of one table into another table using subquery with WHERE clause.
Let's take Old_Employee and New_Employee tables. The Old_Employee and New_Employee table contain the same number of columns. But both the tables contain different records.

## Table: Old_Employee

| Emp_ID | Emp_Name | Emp_Salary | Address |
|---|---|---|---|
| 1001 | Akhil | 50000 | Agra |
| 1002 | Balram | 25000 | Delhi |
| 1003 | Bheem | 45000 | Gurgaon |
| 1004 | Chetan | 60000 | Noida |
| 1005 | Diksha | 30000 | Agra |
| 1006 | Raman | 50000 | Ghaziabad |
| 1007 | Sheetal | 35000 | Delhi |

The New_Employee contains the details of new employees.

If you want to move the details of those employees whose salary is greater than 40000 from the Old_Employee table to the New_Employee table.

Then for this issue, you must type the following query in SQL:

INSERT INTO New_Employee SELECT * FROM Old_Employee WHERE Emp_Salary > 40000;

Now, you can check the details of the updated New_Employee table by using the following SELECT query:

SELECT * FROM New_Employee;

**Table: New_Employee**

**Output:**

| Emp_ID | Emp_Name | Emp_Salary | Address |
|---|---|---|---|
| 1008 | Sumit | 50000 | Agra |
| 1009 | Akash | 55000 | Delhi |
| 1010 | Devansh | 65000 | Gurgaon |
| 1001 | Akhil | 50000 | Agra |
| 1003 | Bheem | 45000 | Gurgaon |
| 1004 | Chetan | 60000 | Noida |
| 1006 | Raman | 50000 | Ghaziabad |

## SQL Server indexes

SQL Server indexes are used to help retrieve data quicker and reduce bottlenecks impacting critical resources. Indexes on a database table serve as a **performance optimization** technique.

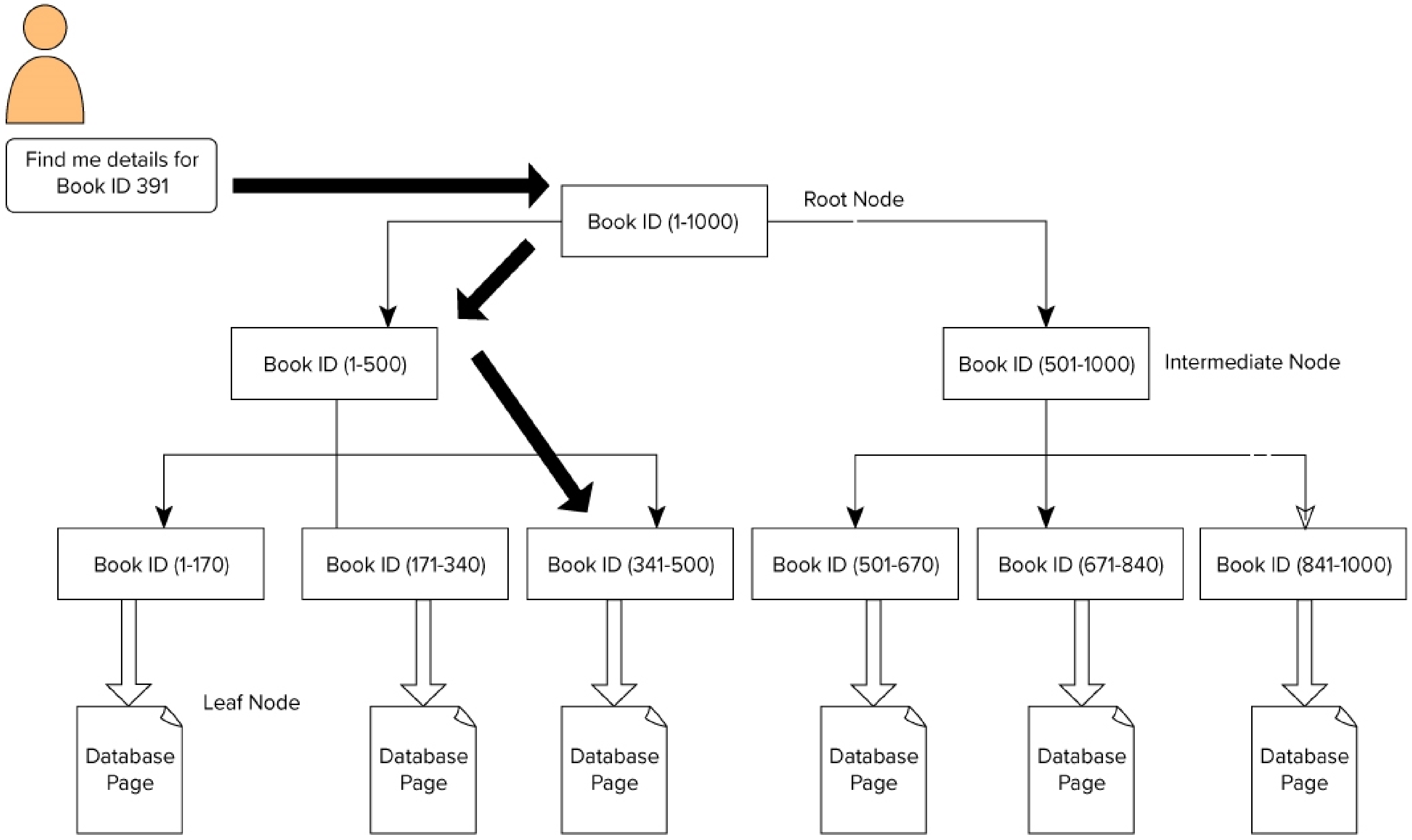
Imagine you visit a city library that has a collection of thousands of books. You’re looking for a specific book, but how will you find it? If you went through each book, in each rack, it could take days to find it. The same applies to a database when you are looking for a record from the millions of rows stored in a table.

A SQL Server index is shaped in a **B-Tree format** that consists of a root node at the top and leaf node at the bottom. For our library books example, a user issues a query to search for a book with the ID 391. In this case, the query engine starts traversing from the root node and moves to the leaf node.

Root Node - > Intermediate node - > Leaf node.

The query engine looks for the reference page in the intermediate level. In this example, the first intermediate node consists of book IDs from 1-500 and the second intermediate node consists of 501-1000.

Based on the intermediate node, the query engine traverses through the B-Tree to look for the corresponding intermediate node and the leaf node. This leaf node can consist of actual data or point to the actual data page based on the index type. In the below image, we see how to traverse the index to look for data using SQL Server indexes. In this case, SQL Server does not have to go through each page, read it and look for a specific book ID content.



Impacts of indexes on SQL Server performance

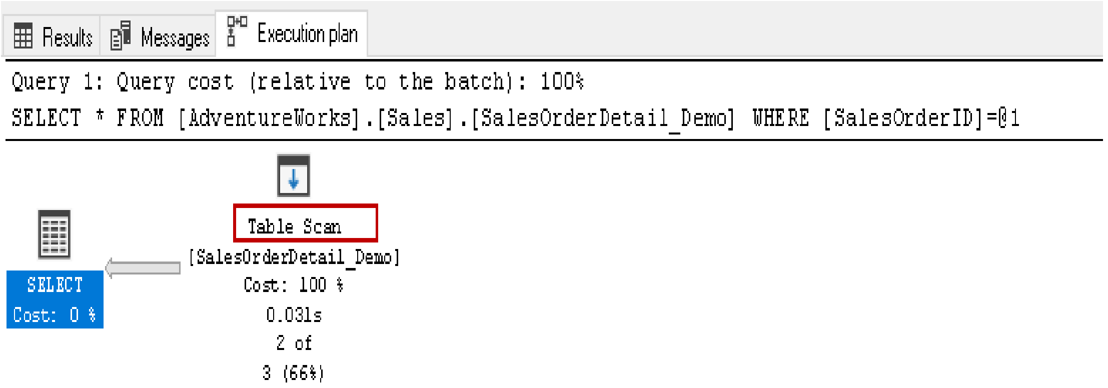
In the previous library example, we examined the potential index performance impacts.
Let's look at the query performance with and without an index.

Suppose we require data for the [SalesOrderID] 56958 from the [SalesOrderDetail_Demo] table.

SELECT *
FROM [AdventureWorks].[Sales].[SalesOrderDetail_Demo]
Where SalesOrderID=56958

This table does not have any indexes on it. A table without any indexes is called a **heap** table in SQL Server.

From here, you would want to run the above select statement and view the actual execution plan. This table has 121317 records in it. It performs a table scan, which means it reads all rows in a table to find the specific [SalesOrderID].



- Now, think of a table with millions or billions of rows. It is not a good practice to traverse through all the records in the table to filter a few rows. In an extensive **online transaction processing (OLTP)** database system, it does not use server resources (CPU, IO, memory) effectively, therefore, the user could face performance issues.

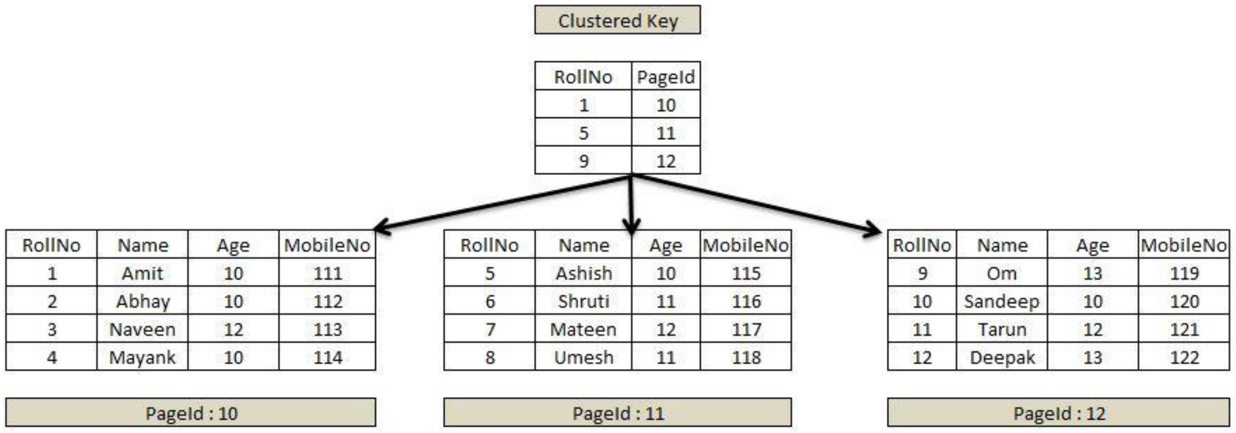
## 1. Clustered Index :

Clustered index is created only when both the following conditions satisfy:

1.The data or file, that you are moving into secondary memory should be in sequential or sorted order.

2. There should be a key value, meaning it can not have repeated values.

In clustered index, index contains **pointer to block but not direct data**.




| RollNo | PageId |
|---|---|
| 1 | 10 |
| 5 | 11 |
| 9 | 12 |

| RollNo | Name | Age | MobileNo |
|---|---|---|---|
| 1 | Amit | 10 | 111 |
| 2 | Abhay | 10 | 112 |
| 3 | Naveen | 12 | 113 |
| 4 | Mayank | 10 | 114 |

| RollNo | Name | Age | MobileNo |
|---|---|---|---|
| 5 | Ashish | 10 | 115 |
| 6 | Shruti | 11 | 116 |
| 7 | Mateen | 12 | 117 |
| 8 | Umesh | 11 | 118 |

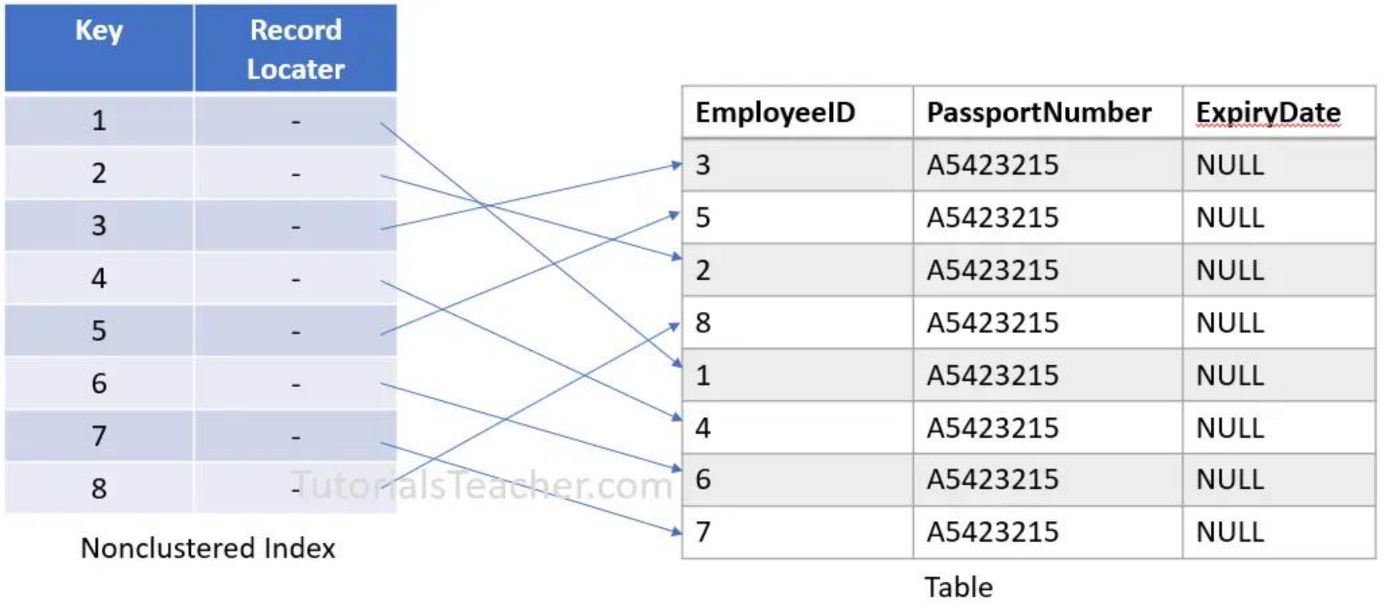
| RollNo | Name | Age | MobileNo |
|---|---|---|---|
| 9 | Om | 13 | 119 |
| 10 | Sandeep | 10 | 120 |
| 11 | Tarun | 12 | 121 |
| 12 | Deepak | 13 | 122 |

## 2. Non-clustered Index :

- Non-Clustered Index is like the index of a book. The index of a book consists of a chapter name and page number, if you want to read any topic or chapter then you can directly go to that page by using index of that book. No need to go through each page of a book.

- The data is stored in one place, and index is stored in another place. Since, the data and non-clustered index is stored separately, then you can have multiple non-clustered index in a table.

In non-clustered index, index contains **<u>the pointer to data</u>**.



Output before applying non-clustered index :

| Roll_No | Name | Gender | Mob_No |
|---|---|---|---|
| 3 | sudhir | male | 9675432890 |
| 4 | afzal | male | 9876543210 |
| 5 | zoya | female | 8976453201 |

Output after applying non-clustered index :

| Name | Row address |
|---|---|
| Afzal | 3452 |
| Sudhir | 5643 |
| zoya | 9876 |

| CLUSTERED INDEX | NON-CLUSTERED INDEX |
| --- | --- |
| Clustered index is faster. | Non-clustered index is slower. |
| Clustered index requires less memory for operations. | Non-Clustered index requires more memory for operations. |
| In clustered index, index is the main data. | In Non-Clustered index, index is the copy of data. |
| A table can have only one clustered index. | A table can have multiple non-clustered index. |
| Clustered index has inherent ability of storing data on the disk. | Non-Clustered index does not have inherent ability of storing data on the disk. |
| Clustered index store pointers to block not data. | Non-Clustered index store both value and a pointer to actual row that holds data. |
| In Clustered index leaf nodes are actual data itself. | In Non-Clustered index leaf nodes are not the actual data itself rather they only contains included columns. |
| In Clustered index, Clustered key defines order of data within table. | In Non-Clustered index, index key defines order of data within index. |
| A Clustered index is a type of index in which table records are physically reordered to match the index. | A Non-Clustered index is a special type of index in which logical order of index does not match physical stored order of the rows on disk. |