**How to Add Column**

ALTER TABLE Employee

ADD LeadTime SmallInt Null

EXEC sp\_help Employees

**Get the Structure of the Table**

EXEC sp\_help Employee

**To alter Column Data Type**

ALTER TABLE Employee

ALTER COLUMN Name varchar (30) NOT NULL

**Computed Column**

CREATE TABLE T (

int1 **int**,

bit1 bit,

varchar1 varchar(3),

dec1 dec(5,2),

cmp1 AS (int1 + bit1)

)

**Derived Column**

SELECT ID

, Name

, Salary \* 2 As DoubleSalary

FROM Employee

**How to Drop a Column**

ALTER TABLE Appointment

DROP COLUMN Resource

**Drop With a Condition**

IF EXISTS(SELECT name FROM sys.tables

WHERE name = 'T')

BEGIN

PRINT 'T already.'

DROP TABLE T\_old

EXEC sp\_rename 'T', 'T\_old'

END

ELSE PRINT 'No T not prese.'

CREATE TABLE T (

c1 bigint,

c2 nvarchar(max)

)

IF EXISTS(SELECT name FROM sys.tables

WHERE name = 'T')

BEGIN

PRINT 'T already.'

DROP TABLE T\_old

EXEC sp\_rename 'T', 'T\_old'

END

ELSE PRINT 'No T not prese.'

SET NOCOUNT ON

drop table T

**Change Column Name**

sp\_rename 'T.c1', 'Test'

here T stands for tablename and c1 stands for column name

**Rename to a table**

sp\_rename T, newEmployee

T is your old and new employee is new tablename

**View the Property of a Table**

EXEC sp\_help Employee

**Create a Temp Table**

Create Table #ASCIIVals (ASCIIValue SmallInt)

But it should always store in Temp database In temporary table section

**Delete and truncate difference and why we should use Truncate**

DECLARE @starttime DateTime

DECLARE @endtime DateTime

DECLARE @totaltime Int

--SET @starttime = GetDate()

--DELETE FROM employee

--SET @endtime = GetDate()

--SET @totaltime = DateDiff(ms, @starttime, @endtime)

--PRINT 'Time to Delete: ' + CONVERT(VarChar(10), @totaltime)

SET @starttime = GetDate()

TRUNCATE TABLE employee

SET @endtime = GetDate()

SET @totaltime = DateDiff(ms, @starttime, @endtime)

PRINT 'Time to Truncate: ' + CONVERT(VarChar(10), @totaltime)

create table employee(id int,name varchar(100))

DECLARE @site\_value INT;

SET @site\_value = 0;

WHILE @site\_value <= 10000

BEGIN

insert into employee values(@site\_value,CONCAT('Bibek',@site\_value))

SET @site\_value = @site\_value + 1;

END;

select \* from employee

**Select Statements**

create table employee(

ID **int**,

name nvarchar (10),

salary **int**,

start\_date datetime,

city nvarchar (10),

region **char** (1))

GO

insert into employee (ID, name, salary, start\_date, city, region)

values (1, 'Jason', 40420, '02/01/94', 'New York', 'W')

insert into employee (ID, name, salary, start\_date, city, region)

values (2, 'Robert',14420, '01/02/95', 'Vancouver','N')

insert into employee (ID, name, salary, start\_date, city, region)

values (3, 'Celia', 24020, '12/03/96', 'Toronto', 'W')

SELECT \* FROM Employee WHERE City LIKE 'V%'

SELECT Name FROM Employee ORDER BY Name

SELECT Name, YEAR(Start\_Date) AS Start\_Year FROM Employee

SELECT TOP 5 ID, Name FROM Employee

SELECT City, COUNT(\*) 'Employees'

FROM Employee

GROUP BY City

HAVING City IS NOT NULL

ORDER BY City

**Using of Between**

CREATE TABLE project (project\_no CHAR(4) NOT NULL,

project\_name CHAR(15) NOT NULL,

budget FLOAT NULL)

insert into project values ('p1', 'Search Engine', 120000.00)

insert into project values ('p2', 'Programming ', 95000.00)

insert into project values ('p3', 'SQL', 186500.00)

GO

select \* from project

select \* from project where budget between 1000 and 120000

select \* from project where budget not between 1000 and 120000

alter name to between

SELECT project\_name, budget FROM project WHERE budget >= 95000 AND budget <= 120000

**IN Clause**

select \* from project where project\_name in ('SQL','PRogramming')

Similar statement to In clause

SELECT \* FROM employee WHERE emp\_no = 2 OR emp\_no = 1 OR emp\_no = 3

select \* from project where project\_name not in ('SQL','PRogramming')

**Null Check**

create table nullcheck(id int,name varchar(50))

insert into nullcheck values(2,null)

select \* from nullcheck where name is not null

select \* from nullcheck where name is null

**ClusterIndex**

Indexes are used to speed-up query process in SQL Server, resulting in high performance. They are similar to textbook indexes. In textbooks, if you need to go to a particular chapter, you go to the index, find the page number of the chapter and go directly to that page. Without indexes, the process of finding your desired chapter would have been very slow.

The same applies to indexes in databases. Without indexes, a DBMS has to go through all the records in the table in order to retrieve the desired results. This process is called table-scanning and is extremely slow. On the other hand, if you create indexes, the database goes to that index first and then retrieves the corresponding table records directly.

There are two types of Indexes in SQL Server:

1. Clustered Index
2. Non-Clustered Index

Clustered Index

A clustered index defines the order in which data is physically stored in a table. Table data can be sorted in only way, therefore, there can be only one clustered index per table. In SQL Server, the primary key constraint automatically creates a clustered index on that particular column.

CREATE DATABASE schooldb

CREATE TABLE student

(

    id INT PRIMARY KEY,

    name VARCHAR(50) NOT NULL,

    gender VARCHAR(50) NOT NULL,

    DOB datetime NOT NULL,

    total\_score INT NOT NULL,

    city VARCHAR(50) NOT NULL

)

Notice here in the “student” table we have set primary key constraint on the “id” column. This automatically creates a clustered index on the “id” column. To see all the indexes on a particular table execute “sp\_helpindex” stored procedure. This stored procedure accepts the name of the table as a parameter and retrieves all the indexes of the table. The following query retrieves the indexes created on student table.

|  |
| --- |
| USE schooldb    EXECUTE sp\_helpindex student |

The above query will return this result:

|  |  |  |
| --- | --- | --- |
| index\_name | index\_description | index\_keys |
| PK\_\_student\_\_3213E83F7F60ED59 | clustered, unique, primary key located on PRIMARY | id |

USE schooldb

INSERT INTO student

VALUES

(6, 'Kate', 'Female', '03-JAN-1985', 500, 'Liverpool'),

(2, 'Jon', 'Male', '02-FEB-1974', 545, 'Manchester'),

(9, 'Wise', 'Male', '11-NOV-1987', 499, 'Manchester'),

(3, 'Sara', 'Female', '07-MAR-1988', 600, 'Leeds'),

(1, 'Jolly', 'Female', '12-JUN-1989', 500, 'London'),

(4, 'Laura', 'Female', '22-DEC-1981', 400, 'Liverpool'),

(7, 'Joseph', 'Male', '09-APR-1982', 643, 'London'),

(5, 'Alan', 'Male', '29-JUL-1993', 500, 'London'),

(8, 'Mice', 'Male', '16-AUG-1974', 543, 'Liverpool'),

(10, 'Elis', 'Female', '28-OCT-1990', 400, 'Leeds');

Now delete the index

CREATE CLUSTERED INDEX IX\_tblStudent\_Gender\_Score

ON student(gender ASC, total\_score DESC)

select \* from student

use the follow command

 This index is created on the “gender” and “total\_score” columns. An index that is created on more than one column is called “composite index”.

## Non-Clustered Indexes

A non-clustered index doesn’t sort the physical data inside the table. In fact, a non-clustered index is stored at one place and table data is stored in another place. This is similar to a textbook where the book content is located in one place and the index is located in another. This allows for more than one non-clustered index per table.

It is important to mention here that inside the table the data will be sorted by a clustered index. However, inside the non-clustered index data is stored in the specified order. The index contains column values on which the index is created and the address of the record that the column value belongs to.

When a query is issued against a column on which the index is created, the database will first go to the index and look for the address of the corresponding row in the table. It will then go to that row address and fetch other column values. It is due to this additional step that non-clustered indexes are slower than clustered indexes.

## Creating a Non-Clustered Index

The syntax for creating a non-clustered index is similar to that of clustered index. However, in case of non-clustered index keyword “NONCLUSTERED” is used instead of “CLUSTERED”. Take a look at the following script.

|  |
| --- |
| use schooldb    CREATE NONCLUSTERED INDEX IX\_tblStudent\_Name  ON student(name ASC) |

The above script creates a non-clustered index on the “name” column of the student table. The index sorts by name in ascending order. As we said earlier, the table data and index will be stored in different places. The table records will be sorted by a clustered index if there is one. The index will be sorted according to its definition and will be stored separately from the table.

# **What is the difference between Clustered and Non-Clustered Indexes in SQL Server?**

August 28, 2017 by [Ben Richardson](https://www.sqlshack.com/author/ben/)

[](https://www.apexsql.com/freesqltools?utm_source=sqlshack&amp;utm_medium=banner&amp;utm_campaign=%5bSQLShack%5dApexSQLfreetoolstop)

Indexes are used to speed-up query process in SQL Server, resulting in high performance. They are similar to textbook indexes. In textbooks, if you need to go to a particular chapter, you go to the index, find the page number of the chapter and go directly to that page. Without indexes, the process of finding your desired chapter would have been very slow.

The same applies to indexes in databases. Without indexes, a DBMS has to go through all the records in the table in order to retrieve the desired results. This process is called table-scanning and is extremely slow. On the other hand, if you create indexes, the database goes to that index first and then retrieves the corresponding table records directly.

There are two types of Indexes in SQL Server:

1. Clustered Index
2. Non-Clustered Index

## Clustered Index

A clustered index defines the order in which data is physically stored in a table. Table data can be sorted in only way, therefore, there can be only one clustered index per table. In SQL Server, the primary key constraint automatically creates a clustered index on that particular column.

Let’s take a look. First, create a “student” table inside “schooldb” by executing the following script, or ensure that your database is [fully backed up](https://www.acuitytraining.co.uk/news-tips/backup-strategies-for-sql-databases/) if you are using your live data:

|  |
| --- |
| CREATE DATABASE schooldb    CREATE TABLE student  (      id INT PRIMARY KEY,      name VARCHAR(50) NOT NULL,      gender VARCHAR(50) NOT NULL,      DOB datetime NOT NULL,      total\_score INT NOT NULL,      city VARCHAR(50) NOT NULL  ) |

Notice here in the “student” table we have set primary key constraint on the “id” column. This automatically creates a clustered index on the “id” column. To see all the indexes on a particular table execute “sp\_helpindex” stored procedure. This stored procedure accepts the name of the table as a parameter and retrieves all the indexes of the table. The following query retrieves the indexes created on student table.

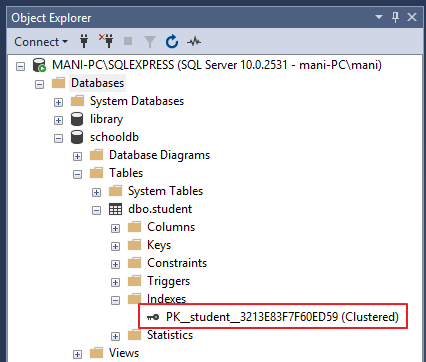
|  |
| --- |
| USE schooldb    EXECUTE sp\_helpindex student |

The above query will return this result:

|  |  |  |
| --- | --- | --- |
| index\_name | index\_description | index\_keys |
| PK\_\_student\_\_3213E83F7F60ED59 | clustered, unique, primary key located on PRIMARY | id |

In the output you can see the only one index. This is the index that was automatically created because of the primary key constraint on the “id” column.

Another way to view table indexes is by going to “Object Explorer-> Databases-> Database\_Name-> Tables-> Table\_Name -> Indexes”. Look at the following screenshot for reference.



This clustered index stores the record in the student table in the ascending order of the “id”. Therefore, if the inserted record has the id of 5, the record will be inserted in the 5th row of the table instead of the first row. Similarly, if the fourth record has an id of 3, it will be inserted in the third row instead of the fourth row. This is because the clustered index has to maintain the physical order of the stored records according to the indexed column i.e. id. To see this ordering in action, execute the following script:

|  |
| --- |
| USE schooldb    INSERT INTO student    VALUES  (6, 'Kate', 'Female', '03-JAN-1985', 500, 'Liverpool'),  (2, 'Jon', 'Male', '02-FEB-1974', 545, 'Manchester'),  (9, 'Wise', 'Male', '11-NOV-1987', 499, 'Manchester'),  (3, 'Sara', 'Female', '07-MAR-1988', 600, 'Leeds'),  (1, 'Jolly', 'Female', '12-JUN-1989', 500, 'London'),  (4, 'Laura', 'Female', '22-DEC-1981', 400, 'Liverpool'),  (7, 'Joseph', 'Male', '09-APR-1982', 643, 'London'),  (5, 'Alan', 'Male', '29-JUL-1993', 500, 'London'),  (8, 'Mice', 'Male', '16-AUG-1974', 543, 'Liverpool'),  (10, 'Elis', 'Female', '28-OCT-1990', 400, 'Leeds'); |

The above script inserts ten records in the student table. Notice here the records are inserted in random order of the values in the “id” column. But because of the default clustered index on the id column, the records are physically stored in the ascending order of the values in the “id” column. Execute the following SELECT statement to retrieve the records from the student table.

|  |
| --- |
| USE schooldb    SELECT \* FROM student |

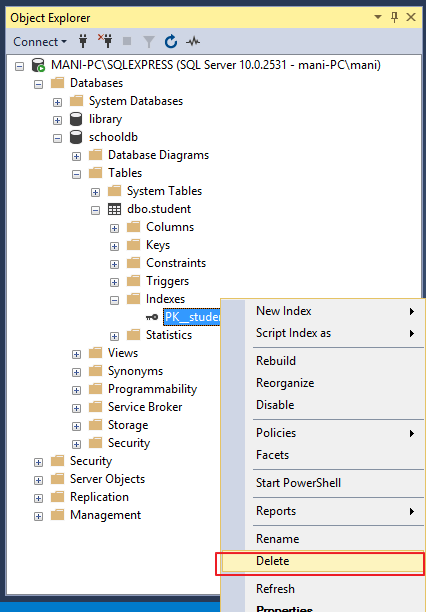
The records will be retrieved in the following order:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | city |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

## Creating Custom Clustered Index

You can create your own custom index as well the default clustered index. To create a new clustered index on a table you first have to delete the previous index.

To delete an index go to “Object Explorer-> Databases-> Database\_Name-> Tables-> Table\_Name -> Indexes”. Right click the index that you want to delete and select DELETE. See the below screenshot.



Now, to create a new clustered Index, execute the following script:

|  |
| --- |
| use schooldb    CREATE CLUSTERED INDEX IX\_tblStudent\_Gender\_Score  ON student(gender ASC, total\_score DESC) |

The process of creating clustered index is similar to a normal index with one exception. With clustered index, you have to use the keyword “CLUSTERED” before “INDEX”.

The above script creates a clustered index named “IX\_tblStudent\_Gender\_Score” on the student table. This index is created on the “gender” and “total\_score” columns. An index that is created on more than one column is called “composite index”.

The above index first sorts all the records in the ascending order of the gender. If gender is same for two or more records, the records are sorted in the descending order of the values in their “total\_score” column. You can create a clustered index on a single column as well. Now if you select all the records from the student table, they will be retrieved in the following order:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | city |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |

## Non-Clustered Indexes

A non-clustered index doesn’t sort the physical data inside the table. In fact, a non-clustered index is stored at one place and table data is stored in another place. This is similar to a textbook where the book content is located in one place and the index is located in another. This allows for more than one non-clustered index per table.

It is important to mention here that inside the table the data will be sorted by a clustered index. However, inside the non-clustered index data is stored in the specified order. The index contains column values on which the index is created and the address of the record that the column value belongs to.

When a query is issued against a column on which the index is created, the database will first go to the index and look for the address of the corresponding row in the table. It will then go to that row address and fetch other column values. It is due to this additional step that non-clustered indexes are slower than clustered indexes.

## Creating a Non-Clustered Index

The syntax for creating a non-clustered index is similar to that of clustered index. However, in case of non-clustered index keyword “NONCLUSTERED” is used instead of “CLUSTERED”. Take a look at the following script.

|  |
| --- |
| use schooldb    CREATE NONCLUSTERED INDEX IX\_tblStudent\_Name  ON student(name ASC) |

The above script creates a non-clustered index on the “name” column of the student table. The index sorts by name in ascending order. As we said earlier, the table data and index will be stored in different places. The table records will be sorted by a clustered index if there is one. The index will be sorted according to its definition and will be stored separately from the table.

**Student Table Data:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | City |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

**IX\_tblStudent\_Name Index Data**

|  |  |
| --- | --- |
| **name** | **Row Address** |
| Alan | Row Address |
| Elis | Row Address |
| Jolly | Row Address |
| Jon | Row Address |
| Joseph | Row Address |
| Kate | Row Address |
| Laura | Row Address |
| Mice | Row Address |
| Sara | Row Address |
| Wise | Row Address |

Notice, here in the index every row has a column that stores the address of the row to which the name belongs. So if a query is issued to retrieve the gender and DOB of the student named “Jon”, the database will first search the name “Jon” inside the index. It will then read the row address of “Jon” and will go directly to that row in the “student” table to fetch gender and DOB of Jon.

## Conclusion

From the discussion we find following differences between clustered and non-clustered indexes.

1. There can be only one clustered index per table. However, you can create multiple non-clustered indexes on a single table.
2. Clustered indexes only sort tables. Therefore, they do not consume extra storage. Non-clustered indexes are stored in a separate place from the actual table claiming more storage space.
3. Clustered indexes are faster than non-clustered indexes since they don’t involve any extra lookup step.

Create a primary key on existing table

ALTER TABLE Employee

ADD CONSTRAINT PK\_EmployeeID

PRIMARY KEY (ID)

The column should have not null constraint

CREATE TABLE employee (emp\_no INTEGER NOT NULL CONSTRAINT prim\_empl PRIMARY KEY,

emp\_fname CHAR(20) NOT NULL,

emp\_lname CHAR(20) NOT NULL,

dept\_no CHAR(4) NULL)

CREATE TABLE project (project\_no CHAR(4) NOT NULL CONSTRAINT prim\_pro PRIMARY KEY,

project\_name CHAR(15) NOT NULL,

budget FLOAT NULL)

CREATE TABLE works\_on1

(emp\_no INTEGER NOT NULL,

project\_no CHAR(4) NOT NULL,

job CHAR (15) NULL,

enter\_date DATETIME NULL,

CONSTRAINT prim\_works1 PRIMARY KEY(emp\_no, project\_no),

CONSTRAINT foreign1\_works1 FOREIGN KEY(emp\_no) REFERENCES employee(emp\_no) ON DELETE CASCADE,

CONSTRAINT foreign2\_works1 FOREIGN KEY(project\_no) REFERENCES project(project\_no) ON UPDATE CASCADE)

Check Constraint

## SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CREATE TABLE Billings

(BillingID INT NOT NULL IDENTITY PRIMARY KEY,

BillingTotal MONEY NOT NULL CHECK (BillingTotal > 0))

GO

INSERT Billings VALUES (-100)

INSERT T (vch1, vch2) VALUES('a','b')

INSERT T (vch1, vch2) VALUES('','b')

INSERT T (vch1, vch2) VALUES('a','')

INSERT T DEFAULT VALUES

CREATE TABLE Billings

(BillingID INT NOT NULL IDENTITY PRIMARY KEY,

BillingTotal MONEY NOT NULL CHECK (BillingTotal >= 0),

PaymentTotal MONEY NOT NULL DEFAULT 0

CHECK (PaymentTotal >= 0))

GO

CREATE TABLE Billings

(BillingID INT NOT NULL IDENTITY PRIMARY KEY,

BillingTotal MONEY NOT NULL,

PaymentTotal MONEY NOT NULL DEFAULT 0,

CHECK ((BillingTotal >= 0) AND (PaymentTotal >= 0)))

GO

ALTER TABLE Customers

WITH NOCHECK

ADD CONSTRAINT CN\_CustomerPhoneNo

CHECK

(Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')

Disabling Constraint

ALTER TABLE MyTable NOCHECK CONSTRAINT FK1\_MyTable