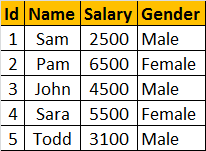
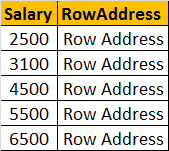
**Indexes in sql server**

**Why indexes?**

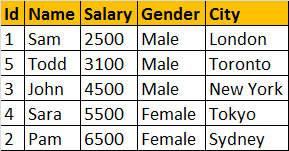
Indexes are used by queries to find data from tables quickly. Indexes are created on tables and views. Index on a table or a view, is very similar to an index that we find in a book.  
  
If you don't have an index in a book, and I ask you to locate a specific chapter in that book, you will have to look at every page starting from the first page of the book.  
  
On, the other hand, if you have the index, you lookup the page number of the chapter in the index, and then directly go to that page number to locate the chapter.  
  
Obviously, the book index is helping to drastically reduce the time it takes to find the chapter.  
  
In a similar way, Table and View indexes, can help the query to find data quickly.  
  
In fact, the existence of the right indexes, can drastically improve the performance of the query. If there is no index to help the query, then the query engine, checks every row in the table from the beginning to the end. This is called as Table Scan. Table scan is bad for performance.  
  
  
  
  
  
  
  
**Index Example:**At the moment, the Employees table, does not have an index on SALARY column.  
  
  
  
**Consider, the following query**  
Select \* from tblEmployee where Salary > 5000 and Salary < 7000  
  
To find all the employees, who has salary **greater than 5000 and less than 7000**, the query engine has to check each and every row in the table, resulting in a table scan, which can adversely affect the performance, especially if the table is large. Since there is no index, to help the query, the query engine performs an entire table scan.  
  
**Now Let's Create the Index to help the query:**Here, we are creating an index on Salary column in the employee table  
CREATE Index IX\_tblEmployee\_Salary   
ON tblEmployee (SALARY ASC)  
  
**The index stores salary of each employee, in the ascending order** as shown below. The actual index may look slightly different.  


**Now, when the SQL server has to execute the same query**, it has an index on the salary column to help this query. Salaries between the range of 5000 and 7000 are usually present at the bottom, since the salaries are arranged in an ascending order. SQL server picks up the row addresses from the index and directly fetch the records from the table, rather than scanning each row in the table. This is called as Index Seek.  
  
**An Index can also be created graphically using SQL Server Management Studio**  
1. In the Object Explorer, expand the Databases folder and then specific database you are working with.  
2. Expand the Tables folder  
3. Expand the Table on which you want to create the index  
4. Right click on the Indexes folder and select New Index  
5. In the New Index dialog box, type in a meaningful name  
6. Select the Index Type and specify Unique or Non Unique Index  
7. Click the Add  
8. Select the columns that you want to add as index key  
9 Click OK  
10. Save the table  
  
**To view the Indexes**: In the object explorer, expand Indexes folder. Alternatively use sp\_helptext system stored procedure. The following command query returns all the indexes on tblEmployee table.  
**Execute sp\_helptext tblEmployee**  
  
**To delete or drop the index:**When dropping an index, specify the table name as well  
Drop Index tblEmployee.IX\_tblEmployee\_Salary

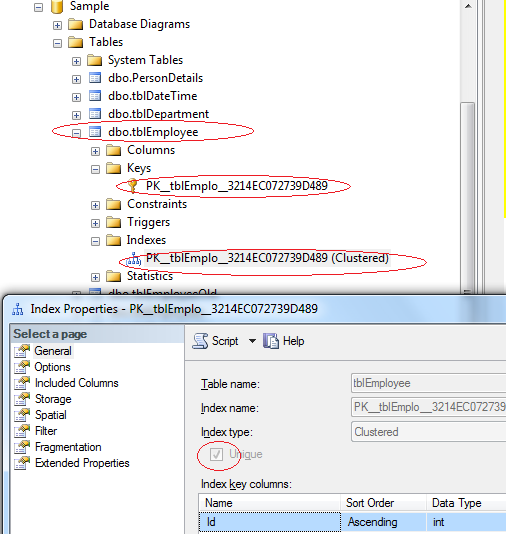
[Email This](https://www.blogger.com/share-post.g?blogID=6082652835152798567&postID=7138284425064405116&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=6082652835152798567&postID=7138284425064405116&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=6082652835152798567&postID=7138284425064405116&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=6082652835152798567&postID=7138284425064405116&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=6082652835152798567&postID=7138284425064405116&target=pinterest)

### Clustered and Non-Clustered indexes

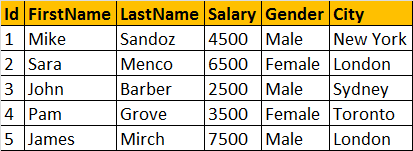
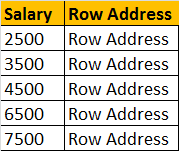
**The following are the different types of indexes in SQL Server**  
1. Clustered  
2. Nonclustered  
3. Unique  
4. Filtered  
5. XML  
6. Full Text  
7. Spatial  
8. Columnstore  
9. Index with included columns  
10. Index on computed columns

**Clustered Index:**  
A clustered index determines the physical order of data in a table. For this reason, a table can have only one clustered index.   
  
**Create tblEmployees table using the script below.**  
CREATE TABLE [tblEmployee]  
(  
 [Id] int Primary Key,  
 [Name] nvarchar(50),  
 [Salary] int,  
 [Gender] nvarchar(10),  
 [City] nvarchar(50)  
)  
  
Note that **Id**column is marked as **primary key**. Primary key, constraint create **clustered indexes automatically** if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.   
  
**To confirm this**, execute sp\_helpindex tblEmployee, which will show a unique clustered index created on the**Id** column.   
  
**Now execute the following insert queries**. Note that, the values for Id column are not in a sequential order.  
Insert into tblEmployee Values(3,'John',4500,'Male','New York')  
Insert into tblEmployee Values(1,'Sam',2500,'Male','London')  
Insert into tblEmployee Values(4,'Sara',5500,'Female','Tokyo')  
Insert into tblEmployee Values(5,'Todd',3100,'Male','Toronto')  
Insert into tblEmployee Values(2,'Pam',6500,'Female','Sydney')  
  
**Execute the following SELECT query**  
Select \* from tblEmployee  
  
**Inspite, of inserting the rows in a random order**, when we execute the select query we can see that all the rows in the table are arranged in an ascending order based on the Id column. This is because a clustered index determines the physical order of data in a table, and we have got a clustered index on the Id column.  
  
**Because of the fact that, a clustered index dictates the physical storage order** of the data in a table, a table can contain only one clustered index. If you take the example of **tblEmployee** table, the data is already arranged by the Id column, and if we try to create another clustered index on the **Name column**, the data needs to be rearranged based on the **NAME column**, which will affect the ordering of rows that's already done based on the ID column.  
  
**For this reason**, SQL server doesn't allow us to create more than one clustered index per table. The following SQL script, raises an error stating 'Cannot create more than one clustered index on table 'tblEmployee'. Drop the existing clustered index PK\_\_tblEmplo\_\_3214EC0706CD04F7 before creating another.'  
Create Clustered Index IX\_tblEmployee\_Name  
ON tblEmployee(Name)  
  
**A clustered index is analogous to a telephone directory**, where the data is arranged by the last name. We just learnt that, a table can have only one clustered index. However, the index can contain multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.  
  
**Let's now create a clustered index on 2 columns**. To do this we first have to drop the existing clustered index on the Id column.   
Drop index tblEmployee.PK\_\_tblEmplo\_\_3214EC070A9D95DB  
  
**When you execute this query**, you get an error message stating 'An explicit DROP INDEX is not allowed on index 'tblEmployee.PK\_\_tblEmplo\_\_3214EC070A9D95DB'. It is being used for PRIMARY KEY constraint enforcement.' We will talk about the role of unique index in the next session. To successfully delete the clustered index, right click on the index in the Object explorer window and select DELETE.  
  
**Now, execute the following CREATE INDEX query**, to create a composite clustered Index on the Gender and Salary columns.  
Create Clustered Index IX\_tblEmployee\_Gender\_Salary  
ON tblEmployee(Gender DESC, Salary ASC)  
  
**Now, if you issue a select query against this table** you should see the data physically arranged, FIRST by Gender in descending order and then by Salary in ascending order. The result is shown below.  
  
  
  
**Non Clustered Index:**  
A nonclustered index is analogous to an index in a textbook. The data is stored in one place, the index in another place. The index will have pointers to the storage location of the data. Since, the nonclustered index is stored separately from the actual data, a table can have more than one non clustered index, just like how a book can have an index by Chapters at the beginning and another index by common terms at the end.  
  
In the index itself, the data is stored in an ascending or descending order of the index key, which doesn't in any way influence the storage of data in the table.   
  
**The following SQL creates a Nonclustered** index on the NAME column on tblEmployee table:  
Create NonClustered Index IX\_tblEmployee\_Name  
ON tblEmployee(Name)  
  
**Difference between Clustered and NonClustered Index:**  
1. **Only one clustered index per table**, where as you can have more than one non clustered index  
2. **Clustered index is faster than a non clustered index**, because, the non-clustered index has to refer back to the table, if the selected column is not present in the index.  
3. **Clustered index determines the storage order of rows in the table**, and hence doesn't require additional disk space, but where as a Non Clustered index is stored seperately from the table, additional storage space is required.

**Unique and Non-Unique Indexes**

**Unique index** is used to enforce uniqueness of key values in the index. Let's understand this with an example.  
  
**Create the Employee table using the script below**  
CREATE TABLE [tblEmployee]  
(  
 [Id] int Primary Key,  
 [FirstName] nvarchar(50),  
 [LastName] nvarchar(50),  
 [Salary] int,  
 [Gender] nvarchar(10),  
 [City] nvarchar(50)  
)  
  
**Since, we have marked Id column**, as the Primary key for this table, a UNIQUE CLUSTERED INDEX gets created on the Id column, with Id as the index key.   
  
**We can verify** this by executing the sp\_helpindex system stored procedure as shown below.  
Execute sp\_helpindex tblEmployee  
  
**Output:**  
  
  
  
**Since, we now have a UNIQUE CLUSTERED INDEX** **on the Id** **column**, any attempt to duplicate the key values, will throw an error stating 'Violation of PRIMARY KEY constraint 'PK\_\_tblEmplo\_\_3214EC07236943A5'. Cannot insert duplicate key in object dbo.tblEmployee'  
  
**Example**: The following insert queries will fail  
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')  
Insert into tblEmployee Values(1,'John', 'Menco',2500,'Male','London')  
  
**Now let's try to drop the Unique Clustered index** on the Id column. This will raise an error stating - 'An explicit DROP INDEX is not allowed on index tblEmployee.PK\_\_tblEmplo\_\_3214EC07236943A5. It is being used for PRIMARY KEY constraint enforcement.'  
Drop index tblEmployee.PK\_\_tblEmplo\_\_3214EC07236943A5  
  
**So this error message proves that**, SQL server internally, uses the UNIQUE index to enforce the uniqueness of values and primary key.  
  
**Expand keys folder in the object explorer window**, and you can see a primary key constraint. Now, expand the indexes folder and you should see a unique clustered index. In the object explorer it just shows the '**CLUSTERED**' word. To, confirm, this is infact an UNIQUE index, right click and select properties. The properties window, shows the UNIQUE checkbox being selected.  
  
  
  
**SQL Server allows us to delete this UNIQUE CLUSTERED INDEX** from the object explorer. so, Right click on the index, and select DELETE and finally, click OK. Along with the UNIQUE index, the primary key constraint is also deleted.  
  
**Now, let's try to insert duplicate values** for the ID column. The rows should be accepted, without any primary key violation error.  
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')  
Insert into tblEmployee Values(1,'John', 'Menco',2500,'Male','London')  
  
So, the UNIQUE index is used to enforce the uniqueness of values and primary key constraint.  
  
**UNIQUENESS is a property of an Index**, and both CLUSTERED and NON-CLUSTERED indexes can be UNIQUE.  
  
**Creating a UNIQUE NON CLUSTERED index** on the FirstName and LastName columns.  
Create Unique NonClustered Index UIX\_tblEmployee\_FirstName\_LastName  
On tblEmployee(FirstName, LastName)  
  
**This unique non clustered index**, ensures that no 2 entires in the index has the same first and last names. There are no major differences between a unique constraint and a unique index.   
  
**In fact, when you add a unique constraint**, a unique index gets created behind the scenes. To prove this, let's add a unique constraint on the city column of the tblEmployee table.  
ALTER TABLE tblEmployee   
ADD CONSTRAINT UQ\_tblEmployee\_City   
UNIQUE NONCLUSTERED (City)  
  
**At this point, we expect a unique constraint to be created**. Refresh and Expand the constraints folder in the object explorer window. The constraint is not present in this folder. Now, refresh and expand the 'indexes' folder. In the indexes folder, you will see a UNIQUE NONCLUSTERED index with name UQ\_tblEmployee\_City.  
  
Also, executing EXECUTE SP\_HELPCONSTRAINT tblEmployee, lists the constraint as a UNIQUE NONCLUSTERED index.  
  
  
  
**So creating a UNIQUE constraint**, actually creates a UNIQUE index. So a UNIQUE index can be created explicitly, using CREATE INDEX statement or indirectly using a UNIQUE constraint. So, when should you be creating a Unique constraint over a unique index.To make our intentions clear, create a unique constraint, when data integrity is the objective. This makes the objective of the index very clear. In either cases, data is validated in the same manner, and the query optimizer does not differentiate between a unique index created by a unique constraint or manually created.  
  
**Note:**  
**1. By default, a PRIMARY KEY constraint**, creates a unique clustered index, where as a UNIQUE constraint creates a unique nonclustered index. These defaults can be changed if you wish to.  
  
**2. A UNIQUE constraint or a UNIQUE index** cannot be created on an existing table, if the table contains duplicate values in the key columns. Obviously, to solve this,remove the key columns from the index definition or delete or update the duplicate values.  
  
**3. By default, duplicate values are not allowed on key columns**, when you have a unique index or constraint. For, example, if I try to insert 10 rows, out of which 5 rows contain duplicates, then all the 10 rows are rejected. However, if I want only the 5 duplicate rows to be rejected and accept the non-duplicate 5 rows, then I can use IGNORE\_DUP\_KEY option. An example of using IGNORE\_DUP\_KEY option is shown below.  
CREATE UNIQUE INDEX IX\_tblEmployee\_City  
ON tblEmployee(City)  
WITH IGNORE\_DUP\_KEY

### Advantages and disadvantages of indexes

Indexes are used by queries to find data quickly. In this part, we will learn about the different queries that can benefit from indexes.  
  
**Create Employees table**  
CREATE TABLE [tblEmployee]  
(  
 [Id] int Primary Key,  
 [FirstName] nvarchar(50),  
 [LastName] nvarchar(50),  
 [Salary] int,  
 [Gender] nvarchar(10),  
 [City] nvarchar(50)  
)  
  
**Insert sample data:**  
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')  
Insert into tblEmployee Values(2,'Sara', 'Menco',6500,'Female','London')  
Insert into tblEmployee Values(3,'John', 'Barber',2500,'Male','Sydney')  
Insert into tblEmployee Values(4,'Pam', 'Grove',3500,'Female','Toronto')  
Insert into tblEmployee Values(5,'James', 'Mirch',7500,'Male','London')  
  
**Create a Non-Clustered Index on Salary Column**  
Create NonClustered Index IX\_tblEmployee\_Salary  
On tblEmployee (Salary Asc)  
  
**Data from tblEmployee table**  
  
  
**NonClustered Index**  
  
  
  
**The following select query benefits from the index on the Salary column**, because the salaries are sorted in ascending order in the index. From the index, it's easy to identify the records where salary is between 4000 and 8000, and using the row address the corresponding records from the table can be fetched quickly.  
Select \* from tblEmployee where Salary > 4000 and Salary < 8000  
  
**Not only, the SELECT statement, even the following DELETE and UPDATE** statements can also benefit from the index. To update or delete a row, SQL server needs to first find that row, and the index can help in searching and finding that specific row quickly.  
Delete from tblEmployee where Salary = 2500  
Update tblEmployee Set Salary = 9000 where Salary = 7500  
  
**Indexes can also help queries**, that ask for sorted results. Since the Salaries are already sorted, the database engine, simply scans the index from the first entry to the last entry and retrieve the rows in sorted order. This avoids, sorting of rows during query execution, which can significantly imrpove the processing time.  
Select \* from tblEmployee order by Salary  
  
**The index on the Salary column**, can also help the query below, by scanning the index in reverse order.  
Select \* from tblEmployee order by Salary Desc  
  
**GROUP BY queries can also benefit from indexes**. To group the Employees with the same salary, the query engine, can use the index on Salary column, to retrieve the already sorted salaries. Since matching salaries are present in consecutive index entries, it is to count the total number of Employees  at each Salary quickly.   
Select Salary, COUNT(Salary) as Total  
from tblEmployee  
Group By Salary  
  
**Diadvantages of Indexes:**  
**Additional Disk Space**: Clustered Index does not, require any additional storage. Every Non-Clustered index requires additional space as it is stored separately from the table.The amount of space required will depend on the size of the table, and the number and types of columns used in the index.  
  
**Insert Update and Delete statements can become slow**: When **DML** (Data Manipulation Language) statements (**INSERT, UPDATE, DELETE**) modifies data in a table, the data in all the indexes also needs to be updated. Indexes can help, to search and locate the rows, that we want to delete, but too many indexes to update can actually hurt the performance of data modifications.  
  
**What is a covering query?**  
**If all the columns** that you have requested in the SELECT clause of query, are present in the index, then there is no need to lookup in the table again. The requested columns data can simply be returned from the index.  
  
**A clustered index**, always covers a query, since it contains all of the data in a table. A composite index is an index on two or more columns. Both clustered and nonclustered indexes can be composite indexes. To a certain extent, a composite index, can cover a query.