**Whether, you create a database graphically using the designer or, using a query, the following 2 files gets generated.**  
.MDF file - (Master Data File) = Data File (Contains actual data)  
.LDF file - Transaction Log file (Used to recover the database)

You cannot drop a database, if it is currently in use. You get an error stating - Cannot drop database "NewDatabaseName" because it is currently in use. So, if other users are connected, you need to put the database in single user mode and then drop the database.  
Alter Database DatabaseName Set SINGLE\_USER With Rollback Immediate

[What is the difference between varchar and nvarchar](http://stackoverflow.com/questions/144283/what-is-the-difference-between-varchar-and-nvarchar)

An nvarchar column can store any Unicode data. A varchar column is restricted to an 8-bit codepage. Some people think that varchar should be used because it takes up less space. I believe this is not the correct answer. Codepage incompatabilities are a pain, and Unicode is the cure for codepage problems. With cheap disk and memory nowadays, there is really no reason to waste time mucking around with code pages anymore.

All modern operating systems and development platforms use Unicode internally. By using nvarchar rather than varchar, you can avoid doing encoding conversions every time you read from or write to the database. Conversions take time, and are prone to errors. And recovery from conversion errors is a non-trivial problem.

**Foreign keys**

are used to enforce **database integrity**. In layman's terms, A **foreign key** in one table points to a **primary key** in another table. The foreign key constraint prevents invalid data form being inserted into the foreign key column. The values that you enter into the foreign key column, has to be one of the values contained in the table it points to.

**The general formula is here**  
**Alter table ForeignKeyTable add constraintForeignKeyTable\_ForiegnKeyColumn\_FK**  
**FOREIGN KEY (ForiegnKeyColumn) references PrimaryKeyTable (PrimaryKeyColumn)**

**Default Constraint.**

A column default can be specified using Default constraint. The default constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified, including NULL.

**Cascading referential integrity constraint**

allows to define the actions Microsoft SQL Server should take when a user attempts to delete or update a key to which an existing foreign keys points.

**options when setting up Cascading referential integrity constraint**  
**1. No Action**: This is the default behaviour. No Action specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE or UPDATE is rolled back.  
  
  
**2. Cascade**: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted or updated.  
  
  
**3. Set NULL**: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to NULL.    
  
  
**4. Set Default**: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to default values.

**CHECK constraint** is used to **limit the range of the values**, that can be entered for a column.  
  
  
Let's say, we have an integer AGE column, in a table. The AGE in general cannot be less than ZERO and at the same time cannot be greater than 150. But, since AGE is an integer column it can accept negative values and values much greater than 150.

ADD CONSTRAINT CK\_Person\_Age CHECK ( Age >=0 and Age <= 150)

The Check Constraint is always a Boolean expression.

### Identity column

If a column is marked as an identity column, then the values for this column are automatically generated, when you insert a new row into the table.

Create Table tblPerson  
(PersonId int Identity(1,1) Primary Key,  
Name nvarchar(20))

 if you mark a column as an Identity column, you dont have to explicitly supply a value for that column when you insert a new row. The value is automatically calculated and provided by SQL server. So, to insert a row into tblPerson table, just provide value for Name column.  
Insert into tblPerson values ('Todd')

To explicitly supply a value for identity column  
**1.** First turn on identity insert - SET Identity\_Insert tblPerson ON  
2. In the insert query specify the column list  
    Insert into tblPerson(PersonId, Name) values(2, 'John')

* **SCOPE\_IDENTITY()** - returns the last identity value that is **CREATED** in the same session(Connection) and in the same scope.
* **@@IDENTITY** - returns the last identity value that is **CREATED** in the same session(Connection) and across any scope.
* **IDENT\_CURRENT('TableName')** - returns the last identity value that is **CREATED** for a specific table across any session(Connection) and any scope.

### Unique key constraint

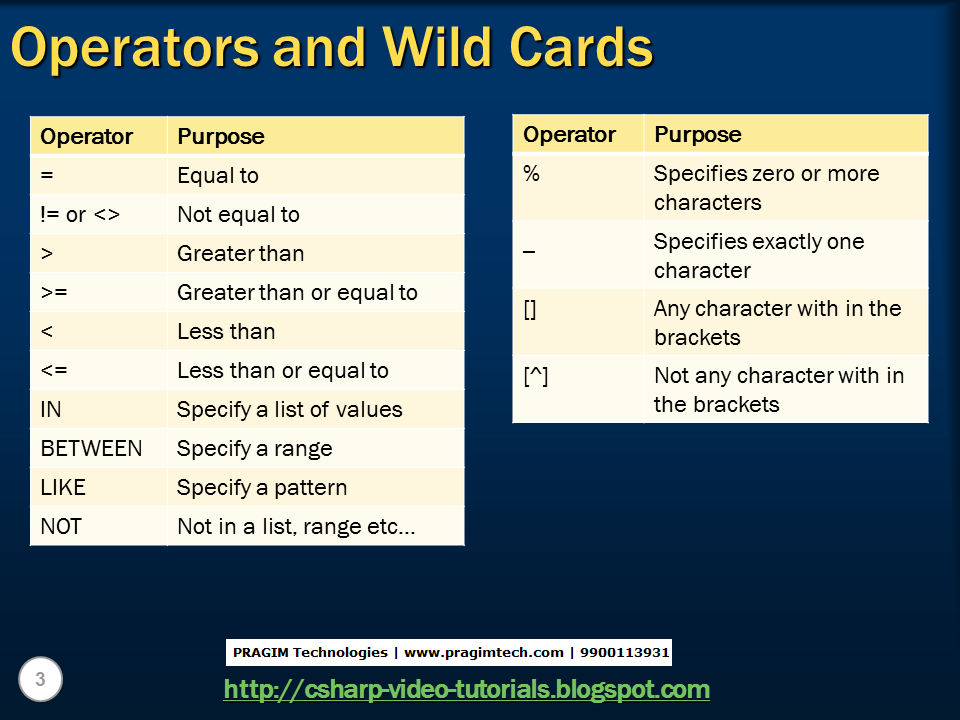
We use UNIQUE constraint to enforce uniqueness of a column i.e the column shouldn't allow any duplicate values. We can add a Unique constraint thru the designer or using a query.

**Both primary key and unique key are used to enforce, the uniqueness of a column. So, when do you choose one over the other?**  
A table can have, only one primary key. If you want to enforce uniqueness on 2 or more columns, then we use unique key constraint.

**What is the difference between Primary key constraint and Unique key constraint? This question is asked very frequently in interviews.**  
**1.** A table can have only one primary key, but more than one unique key  
**2.** Primary key does not allow nulls, where as unique key allows one null

Alter Table Table\_Name  
Add Constraint Constraint\_Name Unique(Column\_Name)

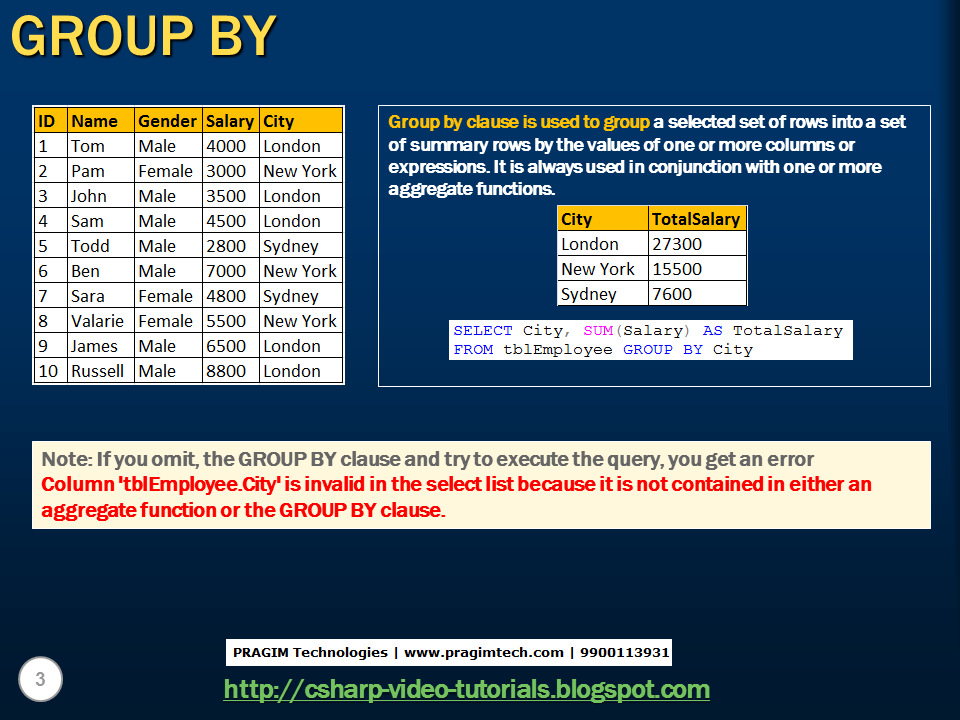
**Where Operators...**

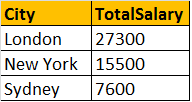


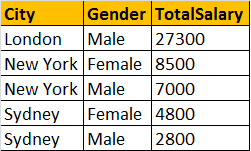
Where **clause** is used to filter data.

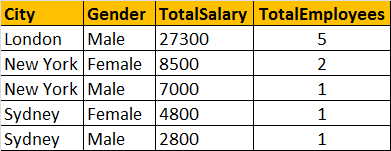
**Group by** clause is used to group a selected set of rows into a set of summary rows by the values of one or more columns or expressions. It is always used in conjunction with one or more aggregate functions.

SQL Server aggregate functions Examples etc...  
1. Count()  
2. Sum()  
3. avg()  
4. Min()  
5. Max()



I want an sql query, which gives total salaries paid by City. The output should be as shown below.   
   
  
**Query for retrieving total salaries by city**:   
We are applying SUM() aggregate function on Salary column, and grouping by city column. This effectively adds, all salaries of employees with in the same city.  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**Group by City**

Now, I want an sql query, which gives total salaries by City, by gender. The output should be as shown below.  
  
  
  
**Query for retrieving total salaries by city and by gender**: It's possible to group by multiple columns. In this query, we are grouping first by city and then by gender.   
**Select City, Gender, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**group by City, Gender**

Now, I want an sql query, which gives total salaries and total number of employees by City, and by gender. The output should be as shown below.   
  


**Query for retrieving total salaries and total number of employees by City, and by gender**: The only difference here is that, we are using Count() aggregate function.  
**Select City, Gender, SUM(Salary) as TotalSalary,   
COUNT(ID) as TotalEmployees**  
**from tblEmployee**  
**group by City, Gender**

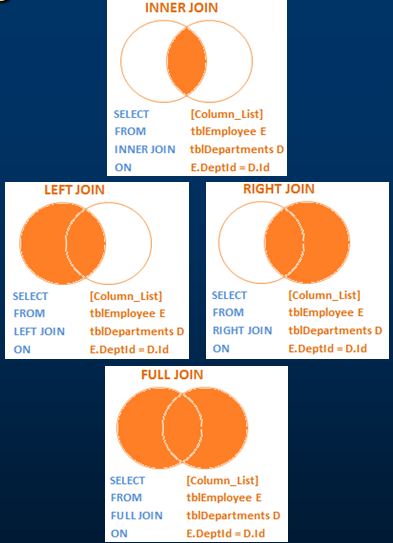
**Filtering Groups:**  
WHERE clause is used to filter rows before aggregation, where as HAVING clause is used to filter groups after aggregations. The following 2 queries produce the same result.

Filtering rows using WHERE clause, before aggrgations take place:  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**Where City = 'London'**  
**group by City**  
  
Filtering groups using HAVING clause, after all aggrgations take place:  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**group by City**  
**Having City = 'London'**

**Difference between WHERE and HAVING clause:**  
1. WHERE clause can be used with - Select, Insert, and Update statements, where as HAVING clause can only be used with the Select statement.  
2. WHERE filters rows before aggregation (GROUPING), where as, HAVING filters groups, after the aggregations are performed.  
3. Aggregate functions cannot be used in the WHERE clause, unless it is in a sub query contained in a HAVING clause, whereas, aggregate functions can be used in Having clause.

**JOINS.**  
**In SQL server, there are different types of JOINS.**  
1. CROSS JOIN  
2. INNER JOIN   
3. OUTER JOIN

**General Formula for Joins**  
SELECT      ColumnList  
FROM           LeftTableName  
JOIN\_TYPE  RightTableName  
ON                 JoinCondition



\*In reality, INNER JOIN and LEFT JOIN are extensively used.

**JOIN or INNER JOIN**

returns only the matching rows between both the tables. Non matching rows are eliminated.

**LEFT JOIN or LEFT OUTER JOIN**

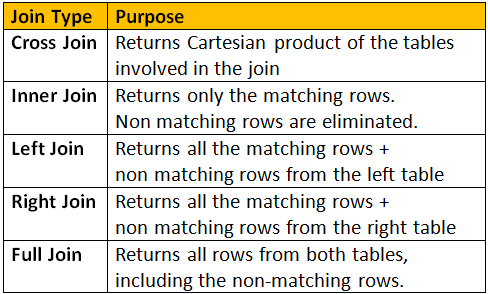
returns all the matching rows + non matching rows from the left table.

**RIGHT JOIN** or **RIGHT OUTER JOIN**

returns all the matching rows + non matching rows from the right table.

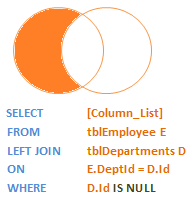
**FULL JOIN**

returns all rows from both the left and right tables, including the non matching rows.

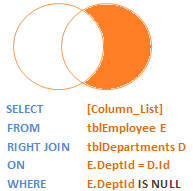


**Advanced or intelligent joins in SQL Server**

1. Retrieve only the non matching rows from the left table



2. Retrieve only the non matching rows from the right table



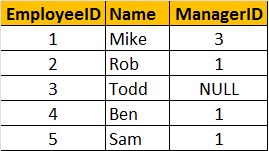
3. Retrieve only the non matching rows from both the left and right table

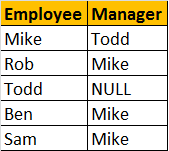


**SELF JOINS**

joining a table with itself is called as **SELF JOIN**. SELF JOIN is not a different type of JOIN. It can be classified under any type of JOIN - INNER, OUTER or CROSS Joins. The above query is, LEFT OUTER SELF Join.

**example...**





A MANAGER is also an EMPLOYEE. Both the, EMPLOYEE and MANAGER rows, are present in the same table. Here we are joining tblEmployee with itself using different alias names, E for Employee and M for Manager. We are using LEFT JOIN, to get the rows with ManagerId NULL. You can see in the output TODD's record is also retrieved, but the MANAGER is NULL. If you replace LEFT JOIN with INNER JOIN, you will not get TODD's record.  
Select E.Name as Employee, M.Name as Manager  
from tblEmployee E  
Left Join tblEmployee M  
On E.ManagerId = M.EmployeeId

### Different ways to replace NULL

**Replacing NULL value using ISNULL() function:** We are passing 2 parameters to IsNULL() function. If M.Name returns NULL, then 'No Manager' string is used as the replacement value.  
SELECT E.Name as Employee, ISNULL(M.Name,'No Manager') as Manager  
FROM tblEmployee E  
LEFT JOIN tblEmployee M  
ON E.ManagerID = M.EmployeeID  
  
**Replacing NULL value using CASE Statement:**  
SELECT E.Name as Employee, CASE WHEN M.Name IS NULL THEN 'No Manager'   
 ELSE M.Name END as Manager  
FROM tblEmployee E  
LEFT JOIN tblEmployee M  
ON E.ManagerID = M.EmployeeID  
  
**Replacing NULL value using COALESCE() function:** COALESCE() function, returns the first NON NULL value.  
SELECT E.Name as Employee, COALESCE(M.Name, 'No Manager') as Manager  
FROM tblEmployee E  
LEFT JOIN tblEmployee M  
ON E.ManagerID = M.EmployeeID

### Coalesce() function

According to the MSDN Books online COALESCE() returns the first Non NULL value. Let's understand this with an example.

We are passing **FirstName, MiddleName and LastName** columns as parameters to the COALESCE() function. The COALESCE() function returns the first non null value from the 3 columns.  
**SELECT Id, COALESCE(FirstName, MiddleName, LastName) AS Name**  
**FROM tblEmployee**

### Union and union all

UNION and UNION ALL operators in SQL Server, are used to combine the result-set of two or more SELECT queries.

**Differences between UNION and UNION ALL (Common Interview Question)**  
From the output, it is very clear that, **UNION removes duplicate** rows, where as **UNION ALL does not**. When use UNION, to remove the duplicate rows, sql server has to to do a distinct sort, which is time consuming. For this reason, UNION ALL is much faster than UNION.   
  
**Note:** If you want to see the cost of DISTINCT SORT, you can turn on the estimated query execution plan using CTRL + L.  
  
**Note:** For UNION and UNION ALL to work, the Number, Data types, and the order of the columns in the select statements should be same.

**Difference between JOIN and UNION**  
**JOINS** and **UNIONS** are different things. However, this question is being asked very frequently now. UNION combines the result-set of two or more select queries into a single result-set which includes all the rows from all the queries in the union, where as JOINS, retrieve data from two or more tables based on logical relationships between the tables**. In short, UNION combines rows from 2 or more tables, where JOINS combine columns from 2 or more table.**

### Stored procedures

A stored procedure is group of T-SQL (Transact SQL) statements. If you have a situation, where you write the same query over and over again, you can save that specific query as a stored procedure and call it just by it's name.

To create a stored procedure we use,**CREATE PROCEDURE** or **CREATE PROC** statement.

**To execute the stored procedure**, you can just type the procedure name and press F5, or use EXEC or EXECUTE keywords followed by the procedure name as shown below.  
1. spGetEmployees  
2. EXEC spGetEmployees  
3. Execute spGetEmployees

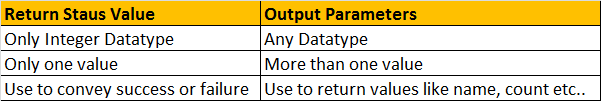
**To encrypt the text of the SP**, use WITH ENCRYPTION option. Once, encrypted, you cannot view the text of the procedure, using sp\_helptext system stored procedure. There are ways to obtain the original text, which we will talk about in a later session.  
Alter Procedure spGetEmployeesByGenderAndDepartment   
@Gender nvarchar(50),  
@DepartmentId int  
WITH ENCRYPTION  
as  
Begin  
  Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId = @DepartmentId  
End

**sp\_help** SP\_Name : View the information about the stored procedure, like parameter names, their datatypes etc. sp\_help can be used with any database object, like tables, views, SP's, triggers etc. Alternatively, you can also press ALT+F1, when the name of the object is highlighted.  
  
**sp\_helptext** SP\_Name : View the Text of the stored procedure  
  
**sp\_depends** SP\_Name : View the dependencies of the stored procedure. This system SP is very useful, especially if you want to check, if there are any stored procedures that are referencing a table that you are abput to drop. sp\_depends can also be used with other database objects like table etc.  
  
**Note:** All parameter and variable names in SQL server, need to have the @symbol.

**What are stored procedure status variables?**  
Whenever, you execute a stored procedure, it returns an integer status variable. Usually, zero indicates success, and non-zero indicates failure.

 I always prefer, using output parameters, over RETURN values because In general, RETURN values are used to indicate success or failure of stored procedure, especially when we are dealing with nested stored procedures.Return a value of 0, indicates success, and any nonzero value indicates failure.

**Difference between return values and output parameters**

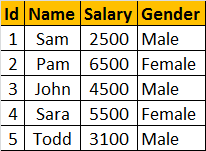


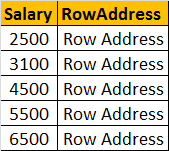
**Advantages of using Stored Procedures over adhoc queries (inline SQL)**

**1. Execution plan retention and reusability** - Stored Procedures are compiled and their execution plan is cached and used again, when the same SP is executed again. Although adhoc queries also create and reuse plan, the plan is reused only when the query is textual match and the datatypes are matching with the previous call. Any change in the datatype or you have an extra space in the query then, a new plan is created.  
  
**2. Reduces network traffic** - You only need to send, EXECUTE SP\_Name statement, over the network, instead of the entire batch of adhoc SQL code.  
  
**3. Code reusability and better maintainability** - A stored procedure can be reused with multiple applications. If the logic has to change, we only have one place to change, where as if it is inline sql, and if you have to use it in multiple applications, we end up with multiple copies of this inline sql. If the logic has to change, we have to change at all the places, which makes it harder maintaining inline sql.  
  
**4. Better Security** - A database user can be granted access to an SP and prevent them from executing direct "select" statements against a table.  This is fine grain access control which will help control what data a user has access to.  
  
**5. Avoids SQL Injection attack** - SP's prevent sql injection attack. [Please watch this video on SQL Injection Attack, for more information.](http://csharp-video-tutorials.blogspot.com/2012/06/sql-injection-attack.html)

### 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.



**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.

### Clustered and Non-Clustered indexes

**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.

**NOTE**: A Clustered index can be a composite index and can be created on more then one column.

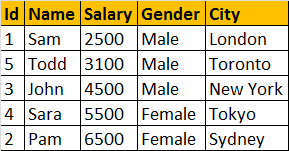
 A Primary key constraint is a **clustered index and is created automatically** if no clustered index already exists on the table and a non-clustered index is not specified when you create the PRIMARY KEY constraint.

**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.

**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

**CREATE** 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.

**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 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.

Create Unique NonClustered Index UIX\_tblEmployee\_FirstName\_LastName

On tblEmployee(FirstName, LastName)

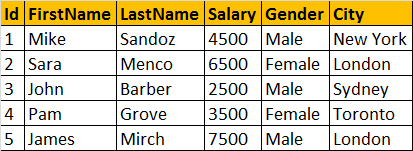
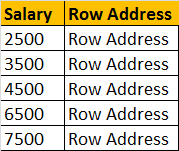
**This unique non clustered index**, ensures that no 2 entires in the index has the same first and last names. We also have learned that a Unique Constraint can be used to enforce the uniqueness of values across one or more columns. **There are no major differences between a unique constraint and a unique index. In fact, when you add a unique constraint**, a unique index gets created behind the scenes.

**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.

**Create a Non-Clustered Index on Salary Column**  
Create NonClustered Index IX\_tblEmployee\_Salary  
On tblEmployee (Salary Asc)

### Advantages and disadvantages of indexes

**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.