**1) ACID Properties in SQL Server**

A - Automicity

C - Consistency

I - Isolation

D - Durability

**Automcity**

All the operations in the transaction must be complete, successfully and be committed, if any one of the operation fails then all the trasactions must be rolled back in their previous state.

**Consistency**

The transaction must be consistent in a state, this means that the transaction must correctly change the state of the system for a particular operation.

**Isolation**

It means that one operation within the transaction cannot see the result of another operation within the transaction.

**Durability**

It means that anything committed to the managed resources must survive or failure and it cannot be done the damages to the resources in case of operation fails.

**2) Normalization**

[**https://www.morpheusdata.com/blog/2015-02-17-pros-cons-db-normalization**](https://www.morpheusdata.com/blog/2015-02-17-pros-cons-db-normalization)

To normalize or not to normalize? Find out when normalization of a database is helpful and when it is not.

TL;DR: When using a relational database, normalization can help keep the data free of errors and can also help ensure that the size of the database doesn't grow large with duplicated data. At the same time, some types of operations can be slower in a normalized environment. So, when should you normalize and when is it better to proceed without normalization?

**What is Normalization?**

*Database normalization is the process of organizing data within a database in the most efficient manner possible*. For example, you likely do not want a username stored in several different tables within your database when you could store it in a single location and point to that user via an ID instead.

By keeping the unchanging user ID in the various tables that need the user, you can always point it back to the appropriate table to get the current username, which is stored in only a single location. Any updates to the username occur only in that place, making the data more reliable.

**What Is Good about Database Normalization?**

A normalized database is advantageous when operations will be write-intensive or when [ACID compliance](http://en.wikipedia.org/wiki/ACID) is required. Some advantages include:

1. Updates run quickly due to no data being duplicated in multiple locations.
2. Inserts run quickly since there is only a single insertion point for a piece of data and no duplication is required.
3. Tables are typically smaller that the tables found in non-normalized databases. This usually allows the tables to fit into the buffer, thus offering faster performance.
4. Data integrity and consistency is an absolute must if the database must be ACID compliant. A normalized database helps immensely with such an undertaking.

**What Are the Drawbacks of Database Normalization?**

A normalized database is not as advantageous under conditions where an application is read-intensive. Here are some of the disadvantages of normalization:

1. Since data is not duplicated, table joins are required. This makes queries more complicated, and thus read times are slower.
2. Since joins are required, indexing does not work as efficiently. Again, this makes read times slower because the joins don't typically work well with indexing.

**What if the Application is Read-Intensive and Write-Intensive?**

In some cases, it isn't as clear that one strategy should be used over the other. Obviously, some applications really need both normalized and non-normalized data to work as efficiently as possible.

In such cases, companies will often use more than one database: a relational data such as MySQL for ACID compliant and write-intensive operations and a NoSQL database such as MongoDB for read-intensive operations on data where duplication is not as big of an issue.

**<http://agiledata.org/essays/dataNormalization.html>**

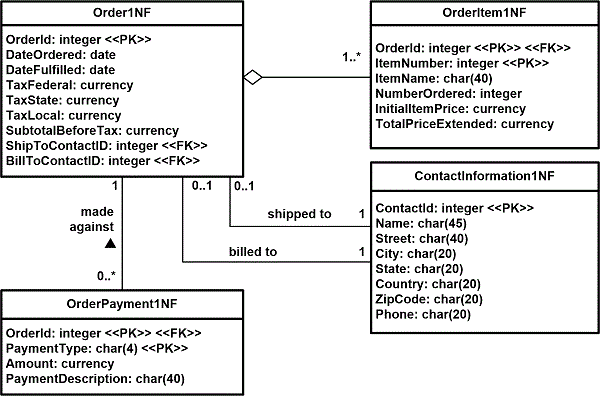
**Table 1. Data Normalization Rules.**

|  |  |
| --- | --- |
| **Level** | **Rule** |
| [First normal form (1NF)](http://agiledata.org/essays/dataNormalization.html#1NF) | An entity type is in 1NF when it contains no repeating groups of data. |
| [Second normal form (2NF)](http://agiledata.org/essays/dataNormalization.html#2NF) | An entity type is in 2NF when it is in 1NF and when all of its non-key attributes are fully dependent on its [primary key](http://www.agiledata.org/essays/keys.html). |
| [Third normal form (3NF)](http://agiledata.org/essays/dataNormalization.html#3NF) | An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the [primary key](http://www.agiledata.org/essays/keys.html). |

**First Normal Form (1NF) (Single Valued Dependency) :** Let’s consider an example. An entity type is in first normal form (1NF) when it contains no repeating groups of data. For example, in Figure 1 you see that there are several repeating attributes in the data Order0NF table – the ordered item information repeats nine times and the contact information is repeated twice, once for shipping information and once for billing information. Although this initial version of orders could work, what happens when an order has more than nine order items? Do you create additional order records for them? What about the vast majority of orders that only have one or two items? Do we really want to waste all that storage space in the database for the empty fields? Likely not. Furthermore, do you want to write the code required to process the nine copies of item information, even if it is only to marshal it back and forth between the appropriate number of objects. Once again, likely not.

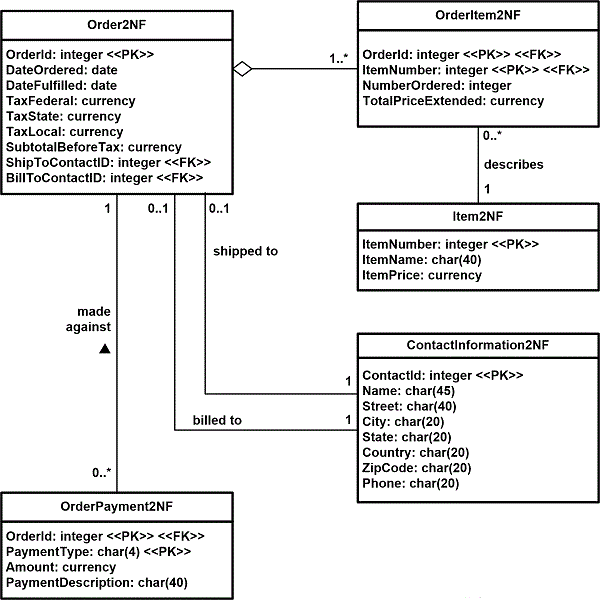


Above figure presents a reworked data schema where the order schema is put in first normal form. The introduction of the OrderItem1NF table enables us to have as many, or as few, order items associated with an order, increasing the flexibility of our schema while reducing storage requirements for small orders (the majority of our business). The ContactInformation1NF table offers a similar benefit, when an order is shipped and billed to the same person (once again the majority of cases) we could use the same contact information record in the database to reduce data redundancy. OrderPayment1NF was introduced to enable customers to make several payments against an order – Order0NF could accept up to two payments, the type being something like “MC" and the description “MasterCard Payment", although with the new approach far more than two payments could be supported, potentially one per payment type. Multiple payments are accepted only when the total of an order is large enough that a customer must pay via more than one approach, perhaps paying some by check and some by credit card.

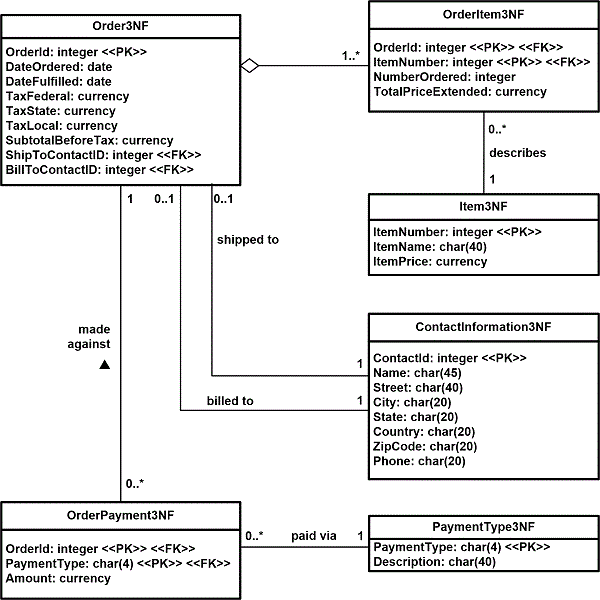


An important thing to notice is the application of primary and foreign keys in the new solution. Order1NF has kept OrderID, the original key of Order0NF, as its primary key. To maintain the relationship back to Order1NF, the OrderItem1NF table includes the OrderID column within its schema, which is why it has the stereotype of FK. When a new table is introduced into a schema, in this case OrderItem1NF, as the result of first normalization efforts it is common to use the primary key of the original table (Order0NF) as part of the primary key of the new table. Because OrderID is not unique for order items, you can have several order items on an order, the column ItemNumber (which is unique to a type of item) was used to form a composite primary key for the OrderItem1NF table. A different approach to keys was taken with the ContactInformation1NF table. The column ContactID, a surrogate key that has no business meaning, was made the primary key.

**Second Normal Form (2NF) (Partial Dependency) :** Although the solution presented in Figure 2 is improved over that of Figure 1, it can be normalized further. Figure 3 presents the data schema of Figure 2 in second normal form (2NF). an entity type is in second normal form (2NF) when it is in 1NF and when every non-key attribute, any attribute that is not part of the primary key, is fully dependent on the primary key. This was definitely not the case with the OrderItem1NF table, therefore we need to introduce the new table Item2NF. The problem with OrderItem1NF is that item information, such as the name and price of an item, do not depend upon an order for that item. For example, if Hal Jordan orders three widgets and Oliver Queen orders five widgets, the facts that the item is called a “widget" and that the unit price is $19.95 is constant. This information depends on the concept of an item, not the concept of an order for an item, and therefore should not be stored in the order items table – therefore the Item2NF table was introduced. OrderItem2NF retained the TotalPriceExtended column, a calculated value that is the number of items ordered multiplied by the price of the item. The value of the SubtotalBeforeTax column within the Order2NF table is the total of the values of the total price extended for each of its order items.



**Third Normal Form (3NF) (Transitive Dependency) :** An entity type is in third normal form (3NF) when it is in 2NF and when all of its attributes are directly dependent on the primary key. A better way to word this rule might be that the attributes of an entity type must depend on all portions of the primary key. In this case there is a problem with the OrderPayment2NF table, the payment type description (such as “Mastercard" or “Check") depends only on the payment type, not on the combination of the order id and the payment type. To resolve this problem the PaymentType3NF table was introduced in Figure 4, containing a description of the payment type as well as a unique identifier for each payment type.



**3) Denormalization :** A denormalized data model is not the same as a data model that has not been normalized, and denormalization should only take place after a satisfactory level of normalization has taken place and that any required constraints and/or rules have been created to deal with the inherent anomalies in the design.

Denormalization is a strategy used on a previously-normalized database to increase performance. In computing, denormalization is the process of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data or by grouping data. It is often motivated by performance or scalability in relational database software needing to carry out very large numbers of read operations. Denormalization should not be confused with Unnormalized form. Databases/tables must first be normalized to efficiently denormalize them.

**4) What is the difference between Clustered and Non-Clustered Indexes in SQL Server**

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:

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

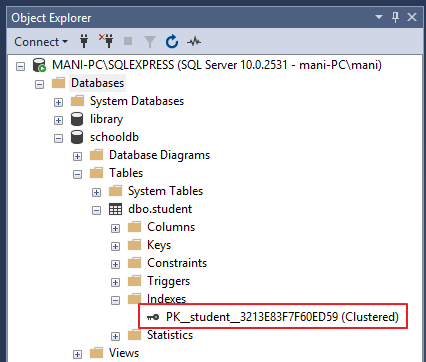
)

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.

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.

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:

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.

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.

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

**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.

**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.

**5) Difference between UNION and UNION ALL**

UNION removes duplicate records (where all columns in the results are the same), UNION ALL does not.

There is a performance hit when using UNION instead of UNION ALL, since the database server must do additional work to remove the duplicate rows, but usually you do not want the duplicates (especially when developing reports).

**UNION Example:**

SELECT 'foo' AS bar UNION SELECT 'foo' AS bar

Result:

+-----+

| **bar** |

+-----+

| foo |

+-----+

1 row in set (0.00 sec)

**UNION ALL example:**

SELECT 'foo' AS bar UNION ALL SELECT 'foo' AS bar

**Result:**

+-----+

| **bar** |

+-----+

| foo |

| foo |

+-----+

2 rows in set (0.00 sec)

**6) What is Cross Join in SQL?**

1. The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN.This kind of result is called as Cartesian Product.
2. *If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.*
3. An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

**7) Natural Join (Not supported in SQL Server 2012 and above)**

A NATURAL JOIN is a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name, types, and lengths in both tables. A NATURAL JOIN can be an INNER join, a LEFT OUTER join, or a RIGHT OUTER join.

One significant difference between INNER JOIN and NATURAL JOIN is the number of columns returned.

Consider:

**TableA**  **TableB**

Column1 Column2 Column1 Column3

1 2 1 3

The INNER JOIN of TableA and TableB on Column1 will return

a.column1 a.column2 b.column1 b.column3

1 2 1 3

SELECT \* FROM TableA INNER JOIN TableB USING (Column1)

SELECT \* FROM TableA INNER JOIN TableB ON TableA.Column1 = TableB.Column1

The NATURAL JOIN of TableA and TableB on Column1 will return:

column1 column2 column3

1 2 3

SELECT \* FROM TableA NATURAL JOIN TableB

The repeated column is avoided.

**8) Join and Inner Join both are same**

Both these joins will give me the same results:

SELECT C.Name FROM Customers C **JOIN** Orders O ON O.CustomerId = C.CustomerId

SELECT C.Name FROM Customers C **INNER** JOIN Orders O ON O.CustomerId = C.CustomerId

**Some Examples:**

**select \* from Customers**

**CustomerId Name**

1 Shree

2 Kalpana

3 Basavaraj

**select \* from Orders**

**OrderId CustomerId**

100 1

200 4

300 3

select \* from Customers **Full JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana NULL NULL

3 Basavaraj 300 3

NULL NULL 200 4

select \* from Customers **Left JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana NULL NULL

3 Basavaraj 300 3

select \* from Customers **Right JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

NULL NULL 200 4

3 Basavaraj 300 3

select \* from Customers **Cross JOIN** Orders

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana 100 1

3 Basavaraj 100 1

1 Shree 200 4

2 Kalpana 200 4

3 Basavaraj 200 4

1 Shree 300 3

2 Kalpana 300 3

3 Basavaraj 300 3

select \* from Customers **Cross JOIN** Orders **where** Customers.customerid=Orders.customerid

***Note :*** *(Works as Inner Join when use where clause in Cross Join)*

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

3 Basavaraj 300 3

select \* from Customers, Orders **(Works as Cross Join)**

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana 100 1

3 Basavaraj 100 1

1 Shree 200 4

2 Kalpana 200 4

3 Basavaraj 200 4

1 Shree 300 3

2 Kalpana 300 3

3 Basavaraj 300 3

select \* from Customers,Orders where Customers.customerid=Orders.customerid

***Note :*** *(Works as Inner Join)*

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

3 Basavaraj 300 3

**Self Join Example : Employee Manager Query**

create table #temp

(

empid int,

mgrid int,

name varchar(50)

)

insert into #temp

values (1,3,'Rahul'), (2,4,'Ajay'), (3,2,'Pankaj'), (4,1,'Sunil')

select a.Empid, a.name as EmpName, b.empid as MgrId, b.name as MgrName

from #temp a left join #temp b

on b.empid=a.mgrid

**Empid EmpName MgrId MgrName**

1 Rahul 3 Pankaj

2 Ajay 4 Sunil

3 Pankaj 2 Ajay

4 Sunil 1 Rahul

**Query to insert the record from #temp to #temp1 which are not in #temp1**

select top 2 \* into #temp1 from #temp

select \* from #temp

**empid mgrid name**

1 3 Rahul

2 4 Ajay

3 2 Pankaj

4 1 Sunil

**select \* from #temp1**

empid mgrid name

1 3 Rahul

2 4 Ajay

insert into #temp1 select \* from (select \* from #temp where empid not in(select empid from #temp1)) tt