Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anamolies.

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **S\_Address** | **Subject\_opted** |
| 401 | Adam | Noida | Bio |
| 402 | Alex | Panipat | Maths |
| 403 | Stuart | Jammu | Maths |
| 404 | Adam | Noida | Physics |

* **Updation Anamoly :** To update address of a student who occurs twice or more than twice in a table, we will have to update **S\_Address** column in all the rows, else data will become inconsistent.
* **Insertion Anamoly :** Suppose for a new admission, we have a Student id(S\_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anamoly.
* **Deletion Anamoly :** If (S\_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

#### First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

**Student Table :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology, Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

**Student Table following 1NF will be :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology |
| Adam | 15 | Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

#### Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

**New Student Table following 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Age** |
| Adam | 15 |
| Alex | 14 |
| Stuart | 17 |

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

**New Subject Table introduced for 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Subject** |
| Adam | Biology |
| Adam | Maths |
| Alex | Maths |
| Stuart | Maths |

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this *transitive functional dependency* should be removed from the table and also the table must be in **Second Normal form**.

Before

**Student\_Detail Table :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Street** | **city** | **State** | **Zip** |

After

**New Student\_Detail Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Zip** |

**Address Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Zip** | **Street** | **city** | **state** |

Transaction Properties

**Atomicity -** This property states that each transaction must be considered as a single unit and must be completed fully or not completed at all. No transaction in the database is left half completed.

**Consistency -** Any transaction should not inject any incorrect or unwanted data into the database. it should maintain the consistency of the database.

**Isolation -**If there are multiple transactions executing simultaneously, then all the transaction should be processed as if they are single transaction. But individual transaction in it should not alter or affect the other transaction. That means each transaction should be executed as if they are independent.

**Durability -** The database should be strong enough to handle any system failure. It should not be working for single transaction alone. It should be able to handle multiple transactions too. If there is any set of insert /update, then it should be able to handle and commit to the database. If there is any failure, the database should be able to recover it to the consistent state.

Suppose if the system crashes after the transaction updates Tom’s address, but before committing the transaction to the database, then the database should be strong enough to recover Tom’s original address. It should not be updated to newer address

Identity - Creates an identity column in a table.

CREATE TABLE new\_employees

(

id\_num int IDENTITY(1,1),

fname varchar (20),

minit char(1),

lname varchar(30)

);

INSERT new\_employees

(fname, minit, lname)

VALUES

('Karin', 'F', 'Josephs');

Primary Key Constraints

CREATE TABLE employee   
( id number(5),   
name char(20),  
dept char(10),  
age number(2),  
salary number(10),  
location char(10),  
CONSTRAINT emp\_id\_pk PRIMARY KEY (id)  
);

Foreign Key Constraint

CREATE TABLE employee  
( id number(5) PRIMARY KEY,  
name char(20),  
dept char(10),  
age number(2),  
mgr\_id number(5) REFERENCES employee(id),  
salary number(10),  
location char(10)   
);

CREATE TABLE employee  
( id number(5),  
name char(20) CONSTRAINT nm\_nn NOT NULL,  
dept char(10),  
age number(2),  
salary number(10),  
location char(10)   
);

Unique Key : A column(s) can have a null value but the values cannot be duplicated.

CREATE TABLE employee  
( id number(5) PRIMARY KEY,  
name char(20),  
dept char(10),  
age number(2),  
salary number(10),  
location char(10) UNIQUE   
);

**SQL Check Constraint :**

This constraint defines a business rule on a column. All the rows must satisfy this rule.

CREATE TABLE employee   
( id number(5) PRIMARY KEY,   
name char(20),   
dept char(10),   
age number(2),   
gender char(1) CHECK (gender in ('M','F')),   
salary number(10),   
location char(10)   
);

* **Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
* **Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data.   
  These Data Manipulation Language commands are: [SELECT](http://beginner-sql-tutorial.com/sql-select-statement.htm), [INSERT](http://beginner-sql-tutorial.com/sql-insert-statement.htm), [UPDATE](http://beginner-sql-tutorial.com/sql-update-statement.htm), and [DELETE](http://beginner-sql-tutorial.com/sql-delete-statement.htm).
* **Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
* **Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

Commit Syntax

begin tran d  
update emp set empName ='D' where empid=11  
commit tran d

Rollback Syntax

begin tran t  
declare @id int;  
set @id=1;  
insert into emp values(@id,'d')  
if(@id<10)  
begin  
print ‘Error Message’;  
rollback tran t;  
end   
else  
begin  
print 'data is inserted'  
end

GRANT SELECT

on employee

to user1

WITH GRANT OPTION; //User 1 and grant select to user 2 on employee table

REVOKE privilege\_name   
ON object\_name   
FROM {user\_name |PUBLIC |role\_name}

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group.

SELECT COUNT (\*) FROM employee WHERE dept = 'Electronics';

SELECT DISTINCT dept FROM employee;

SELECT MAX (salary) FROM employee;

SELECT MIN (salary) FROM employee;

SELECT AVG (salary) FROM employee;

SELECT SUM (salary) FROM employee;

LDF - Log files in SQL Server databases hold all the log information. (Transaction)

*.ndf* is usually recommended to denote secondary data files. It’s also possible to store the secondary data file in a separate physical drive than the one which primary data file is stored.

It points to the other files in the database. Therefore, every database has one primary data file. Also, all the data in the database objects (tables, stored procedures, views, triggers.. etc.) are stored in the primary data files.

The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order.

SELECT name, salary, salary\*1.2 AS new\_salary   
FROM employee   
WHERE salary\*1.2 > 30000   
ORDER BY new\_salary DESC;

The SQL GROUP BY Clause is used along with the group functions to retrieve data grouped according to one or more columns.

SELECT dept, SUM (salary)   
FROM employee   
GROUP BY dept;

# SQL HAVING Clause

Having clause is used to filter data based on the group functions. This is similar to WHERE condition but is used with group functions.

SELECT dept, SUM (salary)   
FROM employee   
GROUP BY dept   
HAVING SUM (salary) > 25000

## Basic Difference

1. Function must return a value but in Stored Procedure it is optional( Procedure can return zero or n values).
2. Functions can have only input parameters for it whereas Procedures can have input/output parameters .
3. Functions can be called from Procedure whereas Procedures cannot be called from Function.

## Advance Difference

1. Procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
2. Procedures can not be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
3. Stored Procedures cannot be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
4. Functions that return tables can be treated as another rowset. This can be used in JOINs with other tables.
5. Inline Function can be though of as views that take parameters and can be used in JOINs and other Rowset operations.
6. Exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
7. We can go for Transaction Management in Procedure whereas we can't go in Function.

System Defined Functions:

* Scalar functions operates on a single value and returns a single value.

upper('dotnet') This will returns upper case of given string means 'DOTNET'

lower('DOTNET') This will returns lower case of given string means 'dotnet'

ltrim(' dotnet') This will remove the spaces from left hand side of 'dotnet' string.

* Aggregate functions operates on a collection of values and returns a single value.

max() -This returns maximum value from a collection of values.

min() - This returns minimum value from a collection of values.

avg() - This returns average of all values in a collection.

count() - This returns no of counts from a collection of values.

User Defined Functions :

* Scalar Function:

1. **Create function fnGetEmpFullName**
2. **(**
3. **@FirstName varchar(50),**
4. **@LastName varchar(50)**
5. **)**
6. **returns varchar(101)**
7. **As**
8. **Begin return (Select @FirstName + ' '+ @LastName);**
9. **end**

## 2. Inline Table-Valued Function

User defined inline table-valued function returns a table variable as a result of actions perform by function. The value of table variable should be derived from a single SELECT statement.

1. **--Create function to get employees**
2. **Create function fnGetEmployee()**
3. **returns Table**
4. **As**
5. **return (Select \* from Employee)**
6. **--Now call the above created function**
7. **Select \* from fnGetEmployee()**

## Multi-Statement Table-Valued Function

User defined multi-statement table-valued function returns a table variable as a result of actions perform by function. In this a table variable must be explicitly declared and defined whose value can be derived from a multiple sql statements.

1. **--Create function for EmpID,FirstName and Salary of Employee**
2. **Create function fnGetMulEmployee()**
3. **returns @Emp Table**
4. **(**
5. **EmpID int,**
6. **FirstName varchar(50),**
7. **Salary int**
8. **)**
9. **As**
10. **begin**
11. **Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;**
12. **--Now update salary of first employee**
13. **update @Emp set Salary=25000 where EmpID=1;**
14. **--It will update only in @Emp table not in Original Employee table**
15. **return**
16. **end**

CTE:

1. This is used to store result of a complex sub query for further use.
2. This is also used to create a recursive query.
3. **;With CTE1(Address, Name, Age)--Column names for CTE, which are optional**
4. **AS**
5. **(**
6. **SELECT Addr.Address, Emp.Name, Emp.Age from Address Addr**
7. **INNER JOIN EMP Emp ON Emp.EID = Addr.EID**
8. **)**
9. **SELECT \* FROM CTE1 --Using CTE**
10. **WHERE CTE1.Age > 50**
11. **ORDER BY CTE1.NAME**

Not stored physically in tempdb

## Temporary Tables

In SQL Server, temporary tables are created at run-time and you can do all the operations which you can do on a normal table. These tables are created inside Tempdb database.

### **Local Temp Table**

Local temp tables are only available to the SQL Server session or connection (means single user) that created the tables. These are automatically deleted when the session that created the tables has been closed. Local temporary table name is stared with single hash ("#") sign.

1. **CREATE TABLE *#LocalTemp***
2. **(**
3. **UserID int,**
4. **Name varchar(50),**
5. **Address varchar(150)**
6. **)**
7. **GO**
8. **insert into *#LocalTemp values ( 1, 'Shailendra','Noida');***
9. **GO**
10. **Select \* from *#LocalTemp***

### **Global Temp Table**

Global temp tables are available to all SQL Server sessions or connections (means all the user). These can be created by any SQL Server connection user and these are automatically deleted when all the SQL Server connections have been closed. Global temporary table name is stared with double hash ("##") sign.

1. **CREATE TABLE *##GlobalTemp***
2. **(**
3. **UserID int,**
4. **Name varchar(50),**
5. **Address varchar(150)**
6. **)**
7. **GO**
8. **insert into *##GlobalTemp values ( 1, 'Shailendra','Noida');***
9. **GO**
10. **Select \* from *##GlobalTemp***

## Table Variable

This acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory. This also allows you to create primary key, identity at the time of Table variable declaration but not non-clustered index.

1. **DECLARE @TProduct TABLE**
2. **(**
3. **SNo INT IDENTITY(1,1),**
4. **ProductID INT,**
5. **Qty INT**
6. **)**
7. **--Insert data to Table variable @Product**
8. **INSERT INTO @TProduct(ProductID,Qty)**
9. **SELECT DISTINCT ProductID, Qty FROM ProductsSales ORDER BY ProductID ASC**
10. **--Select data**
11. **Select \* from @TProduct**
12. **--Next batch**
13. **GO**
14. **Select \* from @TProduct --gives error in next batch**

**Derived Table:**Derived tables are the tables which are created on the fly with the help of the Select statement.

1. **SELECT**\* **FROM**
2. (SELECT e.Emp\_Id,e.EmployeeName,p.Project\_Name FROM dbo.Employee e
3. INNERJOIN
4. dbo.Project p
5. **ON**
6. e.Emp\_Id=p.Project\_Id)Tab
7. **WHERE** Tab.Emp\_Id%2=1

Cursor is a database object to retrieve data from a result set one row at a time, instead of the T-SQL commands that operate on all the rows in the result set at one time.

## Life Cycle of Cursor

## Declare Cursor

A cursor is declared by defining the SQL statement that returns a result set.

## Open

A Cursor is opened and populated by executing the SQL statement defined by the cursor.

## Fetch

When cursor is opened, rows can be fetched from the cursor one by one or in a block to do data manipulation.

## Close

After data manipulation, we should close the cursor explicitly.

## Deallocate

Finally, we need to delete the cursor definition and released all the system resources associated with the cursor.

DECLARE @name VARCHAR(50) -- database name    
DECLARE @path VARCHAR(256) -- path for backup files    
DECLARE @fileName VARCHAR(256) -- filename for backup    
DECLARE @fileDate VARCHAR(20) -- used for file name   
  
SET @path = 'C:\Backup\'    
  
SELECT @fileDate = CONVERT(VARCHAR(20),GETDATE(),112)   
  
DECLARE db\_cursor CURSOR FOR    
SELECT name   
FROM MASTER.dbo.sysdatabases   
WHERE name NOT IN ('master','model','msdb','tempdb')    
  
OPEN db\_cursor     
FETCH NEXT FROM db\_cursor INTO @name     
  
WHILE @@FETCH\_STATUS = 0     
BEGIN     
       SET @fileName = @path + @name + '\_' + @fileDate + '.BAK'    
       BACKUP DATABASE @name TO DISK = @fileName    
  
       FETCH NEXT FROM db\_cursor INTO @name     
END     
  
CLOSE db\_cursor     
DEALLOCATE db\_cursor

Joins

A Join condition is a part of the sql query that retrieves rows from two or more tables

Inner Join

The join that displays only the rows that have a match in both the joined tables is known as inner join.

select e1.Username,e1.FirstName,e1.LastName,e2.DepartmentName from Employee e1 inner join Departments e2 on e1.DepartID=e2.id

Equi Join

Equi join is a special type of join in which we use only equality operator. Hence, when you make a query for join using equality operator, then that join query comes under Equi join.

SELECT \* FROM Employee e1 JOIN Departments e2 ON e1.DepartID = e2.id

##### Left Outer join

Left join displays all the rows from first table and matched rows from second table like that..

Hide   Copy Code

SELECT \* FROM Employee e1 LEFT OUTER JOIN Departments e2

ON e1.DepartID = e2.id

Right outer join displays all the rows of second table and matched rows from first table like that.

Hide   Copy Code

SELECT \* FROM Employee e1 RIGHT OUTER JOIN Departments e2

ON e1.DepartID = e2.id

##### Full outer join

Full outer join returns all the rows from both tables whether it has been matched or not.

Hide   Copy Code

SELECT \* FROM Employee e1 FULL OUTER JOIN Departments e2

ON e1.DepartID = e2.id

#### Cross Join

A cross join that produces Cartesian product of the tables that are involved in the join. The size of a Cartesian product is the number of the rows in the first table multiplied by the number of rows in the second table like this.

Hide   Copy Code

SELECT \* FROM Employee cross join Departments e2

#### Self Join

Joining the table itself called self join. Self join is used to retrieve the records having some relation or similarity with other records in the same table. Here, we need to use aliases for the same table to set a self join between single table and retrieve records satisfying the condition in where clause.

Hide   Copy Code

SELECT e1.Username,e1.FirstName,e1.LastName from Employee e1 inner join Employee e2 on e1.id=e2.DepartID

### **@@IDENTITY**

It returns the last identity value generated for any table in the current session, across all scopes. Suppose we create an insert trigger on table which inserts a row in another table with generate an identity column, then @@IDENTITY returns that identity record which is created by trigger.

### **SCOPE\_IDENTITY**

It returns the last identity value generated for any table in the current session and the current scope.

### **IDENT\_CURRENT**

It returns the last identity value generated for a specific table in any session and any scope.

Table partitioning is a way to divide a large table into smaller, more manageable parts without having to create separate tables for each part. Data in a partitioned table is physically stored in groups of rows called partitions and each partition can be accessed and maintained separately.

Error Handling in SP

USE AdventureWorks2012;

GO

IF OBJECT\_ID('UpdateSales', 'P') IS NOT NULL

DROP PROCEDURE UpdateSales;

GO

CREATE PROCEDURE UpdateSales

  @SalesPersonID INT,

  @SalesAmt MONEY = 0

AS

BEGIN

  BEGIN TRY

    BEGIN TRANSACTION;

      UPDATE LastYearSales

      SET SalesLastYear = SalesLastYear + @SalesAmt

      WHERE SalesPersonID = @SalesPersonID;

    COMMIT TRANSACTION;

  END TRY

  BEGIN CATCH

    IF @@TRANCOUNT > 0

    ROLLBACK TRANSACTION;

    DECLARE @ErrorNumber INT = ERROR\_NUMBER();

    DECLARE @ErrorLine INT = ERROR\_LINE();

    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

    DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

    DECLARE @ErrorState INT = ERROR\_STATE();

    PRINT 'Actual error number: ' + CAST(@ErrorNumber AS VARCHAR(10));

    PRINT 'Actual line number: ' + CAST(@ErrorLine AS VARCHAR(10));

    RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);

  END CATCH

END;

GO

USE AdventureWorks2012;

GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

GO

BEGIN TRANSACTION;

GO

SELECT \*

FROM HumanResources.EmployeePayHistory;

GO

SELECT \*

FROM HumanResources.Department;

GO

COMMIT TRANSACTION;

GO

* **ERROR\_NUMBER():** The number assigned to the error.
* **ERROR\_LINE():** The line number inside the routine that caused the error.
* **ERROR\_MESSAGE():** The error message text, which includes the values supplied for any substitutable parameters, such as times or object names.
* **ERROR\_SEVERITY():** The error’s severity.
* **ERROR\_STATE():** The error’s state number.
* **ERROR\_PROCEDURE():** The name of the stored procedure or trigger that generated the error.

|  |  |
| --- | --- |
| RAISEERROR | THROW |
| SQL Server 7.0 | SQL Server 2012 |
| It always generates new exception and results in the loss of the original exception details | To Re-THROW the original exception caught in the TRY Block, we can just specify the THROW statement without any parameters in the CATCH block. |
| RAISERROR (@ErMessage,               @ErSeverity,               @ErState ) | THROW |
| Statements after RAISEERROR are executed | Statements after THROW are not executed |
| The severity parameter specifies the severity of the exception. | There is no severity parameter. |
| With RAISERROR we can raise the System Exception | With THROW we can’t raise the System Exception. But when it used in CATCH BLOCK it can Re-THROW the system exception |

***NOWAIT will return error if the original table has (transaction) locked on it.***  
***NOLOCK will read the data irrespective of the (transaction) lock on it.***

SELECT ID, Col1

FROM First WITH (NOWAIT)

WHERE ID = 1

When SET ANSI\_NULLS is ON, a SELECT statement that uses WHERE column\_name = **NULL** returns zero rows even if there are null values in column\_name.

Trigger

A trigger is a special kind of a store procedure that executes in response to certain action on the table like insertion, deletion or updation of data. It is a database object which is bound to a table and is executed automatically. You can’t explicitly invoke triggers. The only way to do this is by performing the required action no the table that they are assigned to.

There are three action query types that you use in SQL which are INSERT, UPDATE and DELETE. So, there are three types of triggers and hybrids that come from mixing and matching the events and timings that fire them. Basically, triggers are classified into two main types:

1. After Triggers (For Triggers)
2. Instead Of Triggers

CREATE TRIGGER trgAfterInsert ON [dbo].[Employee\_Test]

FOR INSERT

AS

declare @empid int;

declare @empname varchar(100);

declare @empsal decimal(10,2);

declare @audit\_action varchar(100);

select @empid=i.Emp\_ID from inserted i;

select @empname=i.Emp\_Name from inserted i;

select @empsal=i.Emp\_Sal from inserted i;

set @audit\_action='Inserted Record -- After Insert Trigger.';

insert into Employee\_Test\_Audit

(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@empid,@empname,@empsal,@audit\_action,getdate());

PRINT 'AFTER INSERT trigger fired.'

GO

CREATE TRIGGER trgInsteadOfDelete ON [dbo].[Employee\_Test]

INSTEAD OF DELETE

AS

declare @emp\_id int;

declare @emp\_name varchar(100);

declare @emp\_sal int;

select @emp\_id=d.Emp\_ID from deleted d;

select @emp\_name=d.Emp\_Name from deleted d;

select @emp\_sal=d.Emp\_Sal from deleted d;

BEGIN

if(@emp\_sal>1200)

begin

RAISERROR('Cannot delete where salary > 1200',16,1);

ROLLBACK;

end

else

begin

delete from Employee\_Test where Emp\_ID=@emp\_id;

COMMIT;

insert into Employee\_Test\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@emp\_id,@emp\_name,@emp\_sal,'Deleted -- Instead Of Delete Trigger.',getdate());

PRINT 'Record Deleted -- Instead Of Delete Trigger.'

end

END

GO

Coalesce –

. **It returns the first encountered Not Null Value from employee table.**

select id , name ,coalesce(Ph\_no,Alt\_no,Office\_no) as contact number from employee

## Clustered Index (CI)

A clustered index is something that reorganizes the way records in the table are physically stored. Therefore a table can have only one clustered index. The leaf nodes of a clustered index contain the data pages, by which I mean the key-value pair in the clustered index has the index key and the actual data value. Also remember, a clustered index will be created on a table by default the moment a primary key is created on the table. A clustered index is something like your train ticket B4/24, you know that you need to board coach B4 and sit on seat number 24. So this index physically leads you to your actual seat.

CREATE CLUSTERED INDEX CL\_ID ON SALES(ID);

## Non-Clustered Index (NCI)

A non-clustered index is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk. The leaf node of a non-clustered index does not consist of the data pages but a pointer to it. That goes to say that a non-clustered index can’t survive on its own - it needs a base to live on. A non-clustered index uses a clustered index (if defined) or the heap to build itself.

When a non-clustered index uses the heap, the leaf node (or the pointer) is a physical location of the data. When it uses a clustered index, the leaf node (or the pointer) is the clustered index key value and this key value in turn points to the actual data.

CREATE NONCLUSTERED INDEX NONCI\_PC ON SALES(ProductCode);

### **Row\_Number()**

This function will assign a unique id to each row returned from the query.

DECLARE @Table TABLE (

Col\_Value varchar(2)

)

INSERT INTO @Table (Col\_Value)

VALUES ('A'),('A'),('A'),('B'),('B'),('C'),('C');

SELECT

Col\_Value,

ROW\_NUMBER() OVER (ORDER BY Col\_Value) AS 'RowID'

FROM

@Table;

Output :

| **Col\_Value** | **RowID** |
| --- | --- |
| A | 1 |
| A | 2 |
| A | 3 |
| B | 4 |
| B | 5 |
| C | 6 |
| C | 7 |

Rank() - This function will assign a unique number to each distinct row, but it leaves a gap between the groups. Let me explain with a query, we will use the same query we used above with Rank().

SELECT

Col\_Value,

Rank() OVER (ORDER BY Col\_Value) AS 'RowID'

FROM

@Table;

Output :

| **Col\_Value** | **RowID** |
| --- | --- |
| A | 1 |
| A | 1 |
| A | 1 |
| B | 4 |
| B | 4 |
| C | 6 |
| C | 6 |

### **Dense\_Rank()**

This function is similar to Rank with only difference, this will not leave gaps between groups.

So if we use the same query used for Rank, then:

Hide   Copy Code

SELECT

Col\_Value,

DENSE\_RANK() OVER (ORDER BY Col\_Value) AS 'RowID'

FROM

@Table;

We will get:

| **Col\_Value** | **RowID** |
| --- | --- |
| A | 1 |
| A | 1 |
| A | 1 |
| B | 2 |
| B | 2 |
| C | 3 |
| C | 3 |

select Salesperson.Name from Salesperson

where Salesperson.ID NOT IN(

select Orders.salesperson\_id from Orders, Customer

where Orders.cust\_id = Customer.ID

and Customer.Name = 'Samsonic')

The subquery portion of the SQL above begins after the “NOT IN” statement. The reason that the query above is an uncorrelated subquery is that the subquery can be run ***independently*** of the outer query. Basically, the subquery has no relationship with the outer query.

SELECT \*

FROM Employee Emp1

WHERE (1) = (

SELECT COUNT(DISTINCT(Emp2.Salary))

FROM Employee Emp2

WHERE Emp2.Salary > Emp1.Salary)

 inner subquery uses Emp1.Salary, but the alias Emp1 is created in the outer query. This is why it is called a correlated subquery, because the subquery references a value in it’s WHERE clause (in this case, it uses a column belonging to Emp1) that is used in the outer query.

A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data. The query that defines the view can be from one or more tables or from other views in the current or other databases.

Uses of Views :

* Views can be used as security mechanisms by letting users access data through the view, without granting the users permissions to directly access the underlying base tables of the view.
* Views can also be used when you copy data to and from SQL Server to improve performance and to partition data.

Types of Views

Indexed Views

You index a view by creating a unique clustered index on it. Indexed views can dramatically improve the performance of some types of queries. Indexed views work best for queries that aggregate many rows. They are not well-suited for underlying data sets that are frequently updated.

Partitioned Views  
A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers. This makes the data appear as if from one table. A view that joins member tables on the same instance of SQL Server is a local partitioned view.

System Views  
You can use system views to return information about the instance of SQL Server or the objects defined in the instance. For example, you can query the sys.databases catalog view to return information about the user-defined databases available in the instance.

USE AdventureWorks2012 ;

GO

CREATE VIEW HumanResources.EmployeeHireDate

AS

SELECT p.FirstName, p.LastName, e.HireDate

FROM HumanResources.Employee AS e JOIN Person.Person AS p

ON e.BusinessEntityID = p.BusinessEntityID ;

GO

-- Query the view

SELECT FirstName, LastName, HireDate

FROM HumanResources.EmployeeHireDate

ORDER BY LastName;

-- Modify the view by adding a WHERE clause to limit the rows returned.

ALTER VIEW HumanResources.EmployeeHireDate

AS

SELECT p.FirstName, p.LastName, e.HireDate

FROM HumanResources.Employee AS e JOIN Person.Person AS p

ON e.BusinessEntityID = p.BusinessEntityID

WHERE HireDate < CONVERT(DATETIME,'20020101',101) ;

GO

USE AdventureWorks2012 ;

GO

IF OBJECT\_ID ('HumanResources.EmployeeHireDate', 'V') IS NOT NULL

DROP VIEW HumanResources.EmployeeHireDate;

GO

Just think that someone drops/alters the table dbo.Customer without paying any heed to our view. Now that would leave our view nowhere. Hence schema bind it, this will prevent any such accidents from happening.

ALTER VIEW Bill\_V

WITH SCHEMABINDING

AS

SELECT C.FName

,C.LNme

,P.ProductDesc

,B.DateOfBooking

,P.Price

,B.QTY

,(B.QTY\*P.Price) AS TotalAmountPayable

FROM dbo.BOOKING B

INNER JOIN dbo.PRODUCTS P

ON B.ProductID=P.ProductID

INNER JOIN dbo.Customer C

ON B.CustID=C.CustID;

Now we are licensed to have an Index on this dbo.Bill\_V view.

CREATE UNIQUE CLUSTERED INDEX Bill\_View\_Indx

ON dbo.Bill\_V(Fname,LNme);

#### Features

Are views only meant for reading data in a customized mode? Not really views also facilitate DML (Insert/Update/Delete). But there is a set of rules which needs to be adhered to enable DMLs.

* If you are using a view to insert data, then your view should have a single select and also all the mandatory columns of the “being edited” table must be included in the view unless the table has a default values for all NOT NULL columns of the table.
* Secondly don’t forget, for views with “WITH CHECK” options enabled, it’s important to keep in mind that the data begin inserted qualifies in the WHERE clause of the view and is certain to be selected by the view. Simply put the data you insert is picked up while you select from your view.
* If the view is having joins with more than one table, then most cases chances of modifying capabilities are negligible unless INSTEAD OF Triggers are in place to handle the request.

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX *index\_name*  
ON *table\_name* (*column1*, *column2*, ...);

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX *index\_name*  
ON *table\_name* (*column1*, *column2*, ...);

DROP INDEX table\_name.index\_name;

## Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

CREATE INDEX index\_name

on table\_name (column1, column2);

The following guidelines indicate when the use of an index should be reconsidered:

* Indexes should not be used on small tables.
* Tables that have frequent, large batch update or insert operations.
* Indexes should not be used on columns that contain a high number of NULL values.
* Columns that are frequently manipulated should not be indexed.