**OFFSET- FETCH NEXT CLAUSE IN T-SQL🡪**

The OFFSET-FETCH clause provides you with an option to fetch only a window or page of results from the result set. OFFSET-FETCH can be used only with the ORDER BY clause.

**SYNTAX🡪**

SELECT [ ] , [ ] , [ ] , [ ]

FROM < >

[ORDER BY { order\_by\_expression [ ASC | DESC ] } [ ,...n]

{OFFSET { integer\_constant | offset\_row\_count\_expression } { ROW | ROWS }

[FETCH { FIRST | NEXT } {integer\_constant | fetch\_row\_count\_expression } { ROW | ROWS } ONLY]}

Limitations in Using OFFSET-FETCH🡪

* ORDER BY is mandatory to use OFFSET and FETCH clause.
* OFFSET clause is mandatory with FETCH. You can never use, ORDER BY … FETCH.
* TOP cannot be combined with OFFSET and FETCH in the same query expression.
* The OFFSET/FETCH rowcount expression can be any arithmetic, constant, or parameter expression that will return an integer value. The rowcount expression does not support scalar sub-queries.

|  |  |
| --- | --- |
| Create table tblProducts\_for\_offset  (  Id int primary key identity,  Name nvarchar(25),  [Description] nvarchar(50),  Price int  )  Go  **1 PRODUCT – 1 SHIT – 1 10**  **2 PRODUCT – 2 SHIT – 2 20**  **. . . .**  **. . . .**  **100 PRODUCT – 100 SHIT – 100 1000** | declare @take int  set @take=1  while(@take<=100)  begin  declare @took nvarchar(25)  declare @taken nvarchar(25)  declare @taking int  set @taken='Product '+cast(@take as nvarchar)  set @took='shit '+CAST(@take as nvarchar)  set @taking=@take\*10  insert into tblProducts\_for\_offset(Name, [Description], Price) values(@taken, @took, @taking)  set @take=@take+1  end |

|  |  |
| --- | --- |
| select \*  from tblProducts\_for\_offset  order by Id  offset 10 rows  fetch next 6 rows only | 11 Product 11 shit 11 110  12 Product 12 shit 12 120  13 Product 13 shit 13 130  14 Product 14 shit 14 140  15 Product 15 shit 15 150  16 Product 16 shit 16 160 |
| create proc tblProducts\_for\_offset\_procedure  @page\_no int,  @rows int  as  begin  select \*  from tblProducts\_for\_offset  order by Id  offset (@page\_no-1)\*@rows rows  fetch next @rows rows only  end  tblProducts\_for\_offset\_procedure 3, 10 | 21 Product 21 shit 21 210  22 Product 22 shit 22 220  23 Product 23 shit 23 230  24 Product 24 shit 24 240  25 Product 25 shit 25 250  26 Product 26 shit 26 260  27 Product 27 shit 27 270  28 Product 28 shit 28 280  29 Product 29 shit 29 290  30 Product 30 shit 30 300 |

**View DEPENDENCIES of different entities in SQL SERVER🡪**

To view the dependencies of a table, using SSMS;

1. In **Object Explorer**, expand **Databases**, expand a database, and then expand **Tables**.
2. Right-click a table, and then click **View Dependencies**.
3. In the **Object Dependencies**<object name> dialog box, select either **Objects that depend on** <object name>, or **Objects on which**<object name>**depends**.
4. Select an object in the **Dependencies** grid. The type of object (such as "Trigger" or "Stored Procedure"), appears in the **Type** box.

**Using T-SQL;**

To get the REFERENCING ENTITIES use

**SYS.DM\_SQL\_REFERENCING\_ENTITIES** dynamic management function  
  
To get the REFERENCED ENTITIES use

**SYS.DM\_SQL\_REFERENCED\_ENTITIES** dynamic management function

**Difference between referencing entity and referenced entity**  
A dependency is created between two objects when one object appears by name inside a SQL statement stored in another object. The object which is appearing inside the SQL expression is known as referenced entity and the object which has the SQL expression is known as a referencing entity.  
  
Now, let us say we have a stored procedure and we want to find the all objects that this stored procedure depends on. This can be very achieved using another dynamic management function, sys.dm\_sql\_referenced\_entities.    
  
The following query returns all the referenced entities of the stored procedure sp\_GetEmployeesandDepartments

**Select \* from**

**sys.dm\_sql\_referenced\_entities('dbo.sp\_GetEmployeesandDepartments','Object')**

**Difference between Schema-bound dependency and Non-schema-bound dependency**  
**Schema-bound dependency :**Schema-bound dependency prevents referenced objects from being dropped or modified as long as the referencing object exists  
  
**Example :**A view created with SCHEMABINDING, or a table created with foreign key constraint.  
  
**Non-schema-bound dependency :**A non-schema-bound dependency doesn't prevent the referenced object from being dropped or modified

**SP\_DEPENDS Function in T-SQL🡪**

Displays information about database object dependencies, such as the views and procedures that depend on a table or view, and the tables and views that are depended on by the view or procedure. References to objects outside the current database are not reported.

**sp\_depends;**  
A system stored procedure that returns object dependencies  
For example,

* If you specify a table name as the argument, then the views and procedures that depend on the specified table are displayed
* If you specify a view or a procedure name as the argument, then the tables and views on which the specified view or procedure depends are displayed.

Syntax :Execute sp\_depends 'ObjectName'

Sometime sp\_depends does not report dependencies correctly🡪

1. Delete the table, on which a SP is depending.
2. DON’T DELETE the SP.
3. Now create table, with same name.
4. Sp\_depends will never show you the dependency.

**SEQUENCE in T-SQL🡪**

Creates a sequence object and specifies its properties. A sequence is a user-defined schema bound object that generates a sequence of numeric values according to the specification with which the sequence was created.

SYNTAX:

CREATE SEQUENCE [schema\_name . ] sequence\_name

[ AS [ built\_in\_integer\_type | user-defined\_integer\_type ] ]

[ START WITH <constant> ]

[ INCREMENT BY <constant> ]

[ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]

[ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]

[ CYCLE | { NO CYCLE } ]

[ { CACHE [ <constant> ] } | { NO CACHE } ]

[ ; ]

|  |  |
| --- | --- |
| Property | Description |
| DataType | Built-in integer type (tinyint , smallint, int, bigint, decimal etc...) or user-defined integer type. Default bigint. |
| START WITH | The first value returned by the sequence object |
| INCREMENT BY | The value to increment or decrement by. The value will be decremented if a negative value is specified. |
| MINVALUE | Minimum value for the sequence object |
| MAXVALUE | Maximum value for the sequence object |
| CYCLE | Specifies whether the sequence object should restart when the max value (for incrementing sequence object) or min value (for decrementing sequence object) is reached. Default is NO CYCLE, which throws an error when minimum or maximum value is exceeded. |
| CACHE | Cache sequence values for performance. Default value is CACHE. |

**--Creating the Incrementing Sequence : The following code create a Sequence object that starts with 1 and increments by 1**

create sequence dbo.sequence\_test

as int

start with 1

increment by 1

select next value for dbo.sequence\_test O/P🡪1 2 3 4 5 6 7 . . . . . . . .

**--Retrieving the current sequence value : If you want to see what the current Sequence value before generating the next, use sys.sequences**

select \* from sys.sequences where name='sequence\_test'

**sys.sequences**🡪

|  |  |  |
| --- | --- | --- |
| Column name | Data type | Description |
| <inherited columns> |  | Inherits all columns from [sys.objects](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-objects-transact-sql). |
| start\_value | sql\_variant NOT NULL | The starting value for the sequence object. If the sequence object is restarted by using ALTER SEQUENCE it will restart at this value. When the sequence object cycles it proceeds to the minimum\_value or maximum\_value, not the start\_value. |
| increment | sql\_variant NOT NULL | The value that is used to increment the sequence object after each generated value. |
| minimum\_value | sql\_variant NULL | The minimum value that can be generated by the sequence object. After this value is reached, the sequence object will either return an error when trying to generate more values or restart if the CYCLE option is specified. If no MINVALUE has been specified, this column returns the minimum value supported by the sequence generator’s data type. |
| maximum\_value | sql\_variant NULL | The maximum value that can be generated by the sequence object. After this value is reached the sequence object will either start returning an error when trying to generate more values or restart if the CYCLE option is specified. If no MAXVALUE has been specified this column returns the maximum value supported by the sequence object's data type. |
| is\_cycling | bit NOT NULL | Returns 0 if NO CYCLE has been specified for the sequence object and 1 if CYCLE has been specified. |
| is\_cached | bit NOT NULL | Returns 0 if NO CACHE has been specified for the sequence object and 1 if CACHE has been specified. |
| cache\_size | int NULL | Returns the specified cache size for the sequence object. This column contains NULL if the sequence was created with the NO CACHE option or if CACHE was specified without specifying a cache size. If the value specified by the cache size is larger than the maximum number of values that can be returned by the sequence object, that unobtainable cache size is still displayed. |
| system\_type\_id | tinyint NOT NULL | ID of the system type for sequence object’s data type. |
| user\_type\_id | int NOT NULL | ID of the data type for the sequence object as defined by the user. |
| precision | tinyint NOT NULL | Max precision of the data type. |
| scale | tinyint NOT NULL | Max scale of the type. Scale is returned together with precision to give users complete metadata. Scale is always 0 for sequence objects because only integer types are allowed. |
| current\_value | sql\_variant NOT NULL | The last value obligated. That is, the value returned from the most recent execution of the NEXT VALUE FOR function or the last value from executing the sp\_sequence\_get\_range procedure. Returns the START WITH value if the sequence has never been used. |
| is\_exhausted | bit NOT NULL | 0 indicates that more values can be generated from the sequence. 1 indicates that the sequence object has reached the MAXVALUE parameter and the sequence is not set to CYCLE. The NEXT VALUE FOR function returns an error until the sequence is restarted by using ALTER SEQUENCE. |

**--Alter the Sequence object to reset the sequence value :**

alter sequence dbo.sequence\_test restart with 1

select next value for dbo.sequence\_test O/P- 1 2 3 4 5 . . . . .

**--Using sequence value in an INSERT query :**

CREATE TABLE Employees\_sequence

(

Id INT PRIMARY KEY,

Name NVARCHAR(50),

Gender NVARCHAR(10)

)

select \* from Employees\_sequence

declare @a int

set @a=1

while(@a<5)

begin

insert into Employees\_sequence values(next value for dbo.sequence\_test, 'robin', 'M')

set @a=@a+1

end

O/P🡪

1 robin M

2 robin M

3 robin M

4 robin M

**--Creating the decrementing Sequence : The following code create a Sequence object that starts with 100 and decrements by 1**

create sequence dbo.emp\_shit

as int

start with 100

increment by -1

select next value for emp\_shit O/P🡪100 99 98 97 96 95 94 . . . . . . .

**--Specifying MIN and MAX values for the sequence : Use the MINVALUE and MAXVALUE arguments to specify the MIN and MAX values** **respectively**

create sequence holy\_shit

as int

start with 100

increment by 10

minvalue 100

maxvalue 150

select next value for holy\_shit O/P🡪100 110 120 130 140 150 ERROR

**--error when value goes more than 150**

**--need cycle to solve it**

create sequence proc\_shit

as int

start with 100

increment by 10

minvalue 100

maxvalue 150

cycle O/P🡪 100 110 120 130 140 150 100 110 120 130 140 150 . . . . . . . .

**IDENTITY vs SEQUENCE in T-SQL🡪**

|  |  |
| --- | --- |
| **Identity** | **Sequence** |
| The property is a table column property, that is why, it is tied with ROWS | Sequence is a user defined database object, that is not specified with any object. Means it has **Shared value** |
| CREATE TABLE Employees\_testy  (  Id INT PRIMARY KEY identity(1, 1),  Name NVARCHAR(50),  Gender NVARCHAR(10)  )  CREATE TABLE Employees\_testy\_1  (  Id INT PRIMARY KEY identity(1, 1),  Name NVARCHAR(50),  Gender NVARCHAR(10)  )  o/p🡪 **not shared**   |  |  | | --- | --- | | 1 t 1 M  2 t 2 M  3 t 3 M | 1 t 1 M  2 t 2 M  3 t 3 M | | create sequence dbo.vola\_vola  as int  start with 1  increment by 1  select next value for dbo.vola\_vola  insert into< > values(<>,<>,<>) \*5   |  |  | | --- | --- | | 1 holy M  2 moly M  3 toly M | 4 holy M  5 moly M | |
| To generate next identity value we must insert a row | To generate a SEQUENCE value it is not needed to ADD A ROW, we simple can do this using NEXT VALUE FOR <SN> clause. |
| MAXIMUM Value cant be specified, it depends upon the DATATYPE is used | Maximum value can be specified. |
| Identify property doesn’t have a feature to RESTART value from 1. | Where as sequence value can be restarted using  ALTER SEQUENCE <SN> RESTART WITH 1 |

**GUIDs in T-SQL🡪**

SQL Server UNIQUEIDENTIFIER data type. SQL Server supports a special type named UNIQUEIDENTIFIER, which can be used to store "Globally Unique Identifiers" (GUIDs).

**REMARKS:**

* Is a 16-byte GUID.
* By using the NEWID function.
* By converting from a string constant in the form *xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*, in which each *x* is a hexadecimal digit in the range 0-9 or a-f. For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid uniqueidentifier value.
* Comparison operators can be used with **uniqueidentifier** values
* ordering is not implemented by comparing the bit patterns of the two values
* The only operations that can be performed against a **uniqueidentifier** value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL). No other arithmetic operators can be used.
* All column constraints and properties, except IDENTITY, can be used on the **uniqueidentifier** data type.

**Usages:**

Merge replication and transactional replication with updating subscriptions use **uniqueidentifier** columns to guarantee that rows are uniquely identified across multiple copies of the table.

**ADVANTAGES:**

* a GUID is unique in server, database etc.
* if you need to consolidate multiple tables from multiple server then GUID is used.

**DISADVANTAGES:**

* size 16 bit, whereas int 4bits.
* One of the largest datatypes.
* An index built on GUID is larger & slower
* Hard to read.

**Ex:**

declare @xp uniqueidentifier

select @xp=newid()

select @xp as shit o/p🡪FDF52A15-F4C9-42BB-87B2-4CC79183DCEC

|  |  |  |
| --- | --- | --- |
| **Common part** | **Usadb** | **indiadb** |
| **insert into creat**  **select \* from usaDB.dbo.usa**  **union all**  **select \* from IndiaDB.dbo.india**  --Msg 2627, Level 14, State 1, Line 1  --Violation of PRIMARY KEY constraint 'PK\_creat'. Cannot insert duplicate key in object 'dbo.creat'. The duplicate key value is (1).  --The statement has been terminated.  insert into my\_guid\_lo  select \* from usaDB.dbo.guid\_test  union all  select \* from IndiaDB.dbo.guid\_test  select \* from my\_guid\_lo | **1 pam**  **2 john**  create database usaDB  create table guid\_test  (  id uniqueidentifier primary key default newid(),  name nvarchar(10)  )  **A0D717F1-5262-49A5-AA42-17664249E376 Nick**  **BEF53BF6-1C29-485C-8EC6-5052735CC4A8 john**  **A2FE1CB3-833A-4E57-9BF6-F90DF83FFCE5 Pick** | **1 raj**  **2 sekh**  create database IndiaDB  use IndiaDB  create table guid\_test  (  id uniqueidentifier primary key default newid(),  name nvarchar(10)  )  **0AE08376-2F40-4A4A-877E-62AF8AD5FE11 sara** |
| A0D717F1-5262-49A5-AA42-17664249E376 Nick  BEF53BF6-1C29-485C-8EC6-5052735CC4A8 john  0AE08376-2F40-4A4A-877E-62AF8AD5FE11 sara  A2FE1CB3-833A-4E57-9BF6-F90DF83FFCE5 Pick | | |

**Checking if the GUID is EMPTY or NULL🡪**

**Checking null🡪**

declare @tac uniqueidentifier

set @tac=newid()

if(@tac is null)

begin

print 'its freekin empty'

end

else

begin

print 'its preg'

end

**not null case🡪**

declare @tac uniqueidentifier

set @tac=newid()

if(@tac is null)

begin

set @tac=newid()

print @tac --332105BD-F96E-4451-80D1-5FF70C14F010

end

else

begin

print 'its preg'

end

**getting same guid-->**

declare @tac uniqueidentifier

set @tac='332105BD-F96E-4451-80D1-5FF70C14F010'

if(@tac is null)

begin

set @tac=newid()

print @tac --332105BD-F96E-4451-80D1-5FF70C14F010

end

else

begin

print @tac

end

**altering it using isnull🡪**

declare @tac uniqueidentifier

set @tac='332105BD-F96E-4451-80D1-5FF70C14F010' --always this value

--set @tac=NEWID()

select isnull(@tac, newid()) as b

declare @tac uniqueidentifier

--set @tac='332105BD-F96E-4451-80D1-5FF70C14F010' --always this value

set @tac=NEWID()

select isnull(@tac, newid()) as b ---- different guid

**empty guid🡪**

select cast(CAST(0 as binary) as uniqueidentifier) --00000000-0000-0000-0000-000000000000

SELECT CAST(0x0 AS UNIQUEIDENTIFIER) --00000000-0000-0000-0000-000000000000

**Check if empty guid🡪**

Declare @tac UniqueIdentifier

Set @tac = '00000000-0000-0000-0000-000000000000'

If(@tac = '00000000-0000-0000-0000-000000000000')

Begin

Print 'Guid is Empty' --return

End

Else

Begin

Print 'Guid is not Empty'

End

**DYNAMIC SQL🡪**

**Dynamic SQL** is an enhanced form of Structured Query Language (SQL) that, unlike standard (or static) SQL, facilitates the automatic generation and execution of program statements. This can be helpful when it is necessary to write code that can adjust to varying databases, conditions, or servers.

**What is static and dynamic SQL?**

DynamicSQL is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries. ... StaticSQL provides performance advantages over dynamicSQL because staticSQL is preprocessed, which means the statements are parsed, validated, and optimized only once.

declare @fuu nvarchar(1000)

declare @fun nvarchar(1000)

set @fuu='select \* from Employees\_dynimic '+ 'where FirstName=@firstname and Lastname=@lastname'

set @fun='@firstname nvarchar(100), @lastname nvarchar(100)'

execute sp\_executesql @fuu, @fun, @firstname='Mark',@lastname= 'Hastings'

declare @tok nvarchar(1000)

declare @toka nvarchar(1000)

set @tok='select \* from Employees\_dynimic where Id=@id'

set @toka='@id int'

execute sp\_executesql @tok, @toka, 2

**sp\_executesql takes two pre-defined parameters and any number of user-defined parameters.**  
@statement - The is the first parameter which is mandatory, and contains the SQL statements to execute  
@params - This is the second parameter and is optional. This is used to declare parameters specified in @statement

**1 Mark Hastings Male 60000**

**2 Steve Pound Male 45000**

**3 Ben Hoskins Male 70000**

**4 Philip Hastings Male 45000**

**5 Mary Lambeth Female 30000**

**6 Valarie Vikings Female 35000**

**7 John Stanmore Male 80000**