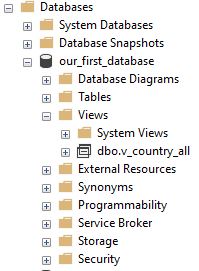
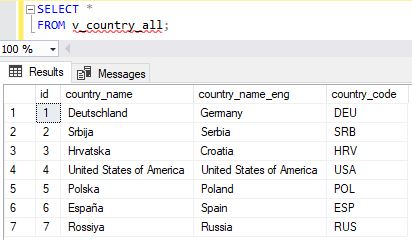
What Are Database Views?

|  |  |
| --- | --- |
| 1  2  3  4 | DROP VIEW IF EXISTS v\_country\_all;  GO  CREATE VIEW v\_country\_all AS    SELECT \* FROM country; |

Similarly to the creating procedures in the previous article, the first line contains the DROP statement (to delete a view if it exists) and after that goes the code that creates a view. After running these statements, the view is created and we can see that in the Object Explorer under Views:



Now, we’ll use this view in the select statement. We’ll go as simple as it’s possible.



You can notice that the result is the same as it would be if we ran the query that is in the view. Also, in our select query, we’ve used the view as we would use any other regular database table.

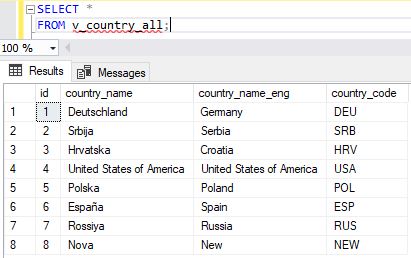
SQL Views – Insert, Updates & Delete

If we can select from the view, this leads to the next question. Can we use the view to insert new rows, update or delete existing? And the answer, in SQL Server, is – “yes”.

So, let’s **insert a new row** using the view we’ve just created:

|  |  |
| --- | --- |
| 1 | INSERT INTO v\_country\_all (country\_name, country\_name\_eng, country\_code) VALUES ('Nova', 'New', 'NEW'); |

The operation completed successfully, and we’ll check if the change in the table is as expected.

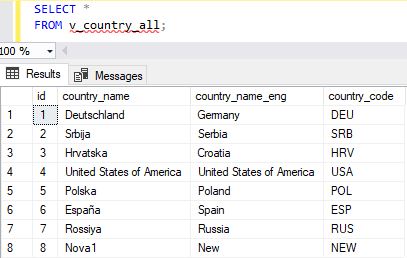


You can notice that we have 1 more line in our table, so the insert using view was performed successfully.

The next thing we’ll try is to **update the existing row** using the view. We’ll update the row we’ve inserted last, using the following statement:

|  |  |
| --- | --- |
| 1  2  3 | UPDATE v\_country\_all SET    country\_name = 'Nova1'  WHERE id = 8; |

You can notice that we’ve updated value for only one column from the view. Once more, we’ll check what happened in the table, selecting from the view:

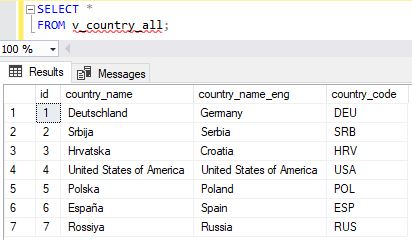


We can notice that the value changes.

The last thing we’ll do using the view is to **delete an existing record**. To do that, we’ll use the following statement:

|  |  |
| --- | --- |
| 1  2  3 | DELETE  FROM v\_country\_all  WHERE id = 8; |

We’ll again check the contents of the table using the combination of select and view.



You can notice that, as expected, the row was deleted.

SQL Views – Advantages & Disadvantages

Like stored procedures, SQL views also have a number of **advantages**. I’ll try to list the most important ones here:

* **Security** – I’ll put security in the first place because, similarly to procedures, you can define who can use a view and how. That same user doesn’t have access to tables used in the view, but only to the view. This way, you can protect sensitive details stored in the table and expose only the ones you want the user to see
* **Easy to use (for the end-user)**– While you might know how to write cool and complex queries, most business users are not interested in that. They just want to get the data. Putting your complex query in the view and allowing business users to use the view, shall hide the complexity of the query and return only the columns they need. You’ll use views as a way how to store your complex code. Also, be aware that you should name your views consistently and logically, so anyone can understand what the view does, simply from its’ name
* **Following business rules & consistency of business logic** – This is related to the previous bullet. If you have specific reports, business users need, you can create a SQL view for every single report. All who need a certain number can simply run this view. If something changes in the reporting requirements, you’ll simply change the view, and all who use it shall immediately feel the effect of that change
* **Use them to make database changes**– Imagine a situation where you want to remove the table, replace it with few tables, or simply changing a table name. In case you do that, there is a great chance you’ll mess up the code somewhere, where this table was used. If you want to prevent that, you could create a view with the same name as the old table had. While this is a fix, this could prove to spare a lot of time
* **Views don’t take space**– Views are used to store your code, not complete tables. Each time you call a view, you’ll run the related query. Therefore, you don’t lose disk space on views

It would be great that we have only advantages, but as it’s usually the case with the most things in life, views also come with some **disadvantages**:

* **Database changes & views** – If you remove an attribute used in the view, the view won’t work. That is the same thing as if you’re trying to run a query using the name of the non-existing column. This is not a big deal if you’re using views only for reporting, because end users will pass the info that their report is not working as expected. In case you’re combining views with insert, update, or delete (some DBMSs allow that) operations, you’ll have a bigger issue
* **Performance** – This could theoretically be a problem because business/end users are usually not aware (and there is no reason why they should be) of what you did. If the query stored in the SQL view is complex and/or not-optimized, it will use a lot of resources and time, and this will lead to all possible issues long queries can cause. We’ll talk more about that later in the series. Still, a business user has no idea of that and could be confused or try to use your view multiple times, etc.

SQL Server date and time data types

Let’s now list all SQL Server date and time data types (starting with the most commonly used):

* **date** – format is YYYY-MM-DD; stores values from 0001-01-01 to 9999-12-31; with the accuracy of 1 day (there is no approximation here because acts same as integer values); uses 3 bytes
* **datetime** –format is YYYY-MM-DD hh:mm:ss[.nnn]; stores values from 1753-01-01 to 9999-12-31; with the accuracy of 0.00333 seconds (please notice we have approximation here); uses 8 bytes
* **time** – format is hh:mm:ss[.nnnnnnn]; stores values from 00:00:00.0000000 to 23:59:59.9999999; with the accuracy of 100 nanoseconds; uses 3 to 5 bytes
* **smalldatetime** – format is YYYY-MM-DD hh:mm:ss; stores values from 1900-01-01 to 2079-06-06; with the accuracy of 1 minute; uses 4 bytes
* **datetime2** –format is YYYY-MM-DD hh:mm:ss[.nnnnnnn]; stores values from 0001-01-01 00:00:00.0000000 to 9999-12-31 23:59:59.9999999; with the accuracy of 100 nanoseconds; uses 6 to 8 bytes
* **datetimeoffset** – format is YYYY-MM-DD hh:mm:ss[.nnnnnnn]; stores values from 0001-01-01 00:00:00.0000000 to 9999-12-31 23:59:59.9999999; with the accuracy of 100 nanoseconds; uses 8 to 10 bytes

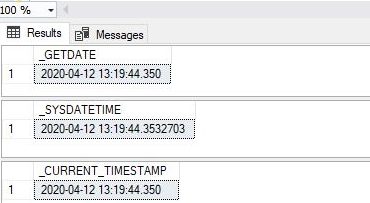
In most cases, you’ll use either **datetime**, either **date**. The remaining 4 types are here if you want to have higher accuracy (**datetime2**, **datetimeoffset**), lower accuracy (**smalldatetime**), or store only time (**time**).

Frequently used SQL Server date and time functions

Similarly to date and time data types, some SQL Server date and time functions are used more often, while some are used rarely. In this part, we’ll check these that are used often, and you can expect you’ll need them in many situations.

If you want to get system values, you’ll use some of the following:

|  |  |
| --- | --- |
| 1  2  3 | SELECT GETDATE() AS \_GETDATE;  SELECT SYSDATETIME() AS \_SYSDATETIME;  SELECT CURRENT\_TIMESTAMP AS \_CURRENT\_TIMESTAMP; |



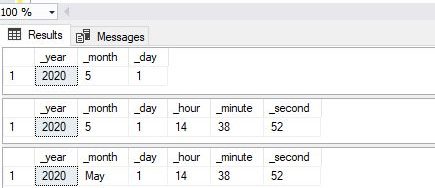
GETDATE() and CURRENT\_TIMESTAMP return the datetime value from the server where SQL Server runs.

SYSDATETIME() returns the same as the previous 2, but with the greater precision, so the result returned is of the datetimeoffset(7).

Besides system date and time functions, we have several other important functions.

The next important set of functions is the one containing functions that return date parts. Let’s take a look at the following statements and their result.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | SELECT  YEAR('2020/05/01 14:38:52') AS \_year,  MONTH('2020/05/01 14:38:52') AS \_month,  DAY('2020/05/01 14:38:52') AS \_day;    SELECT  DATEPART(YEAR, '2020/05/01 14:38:52') AS \_year,  DATEPART(MONTH, '2020/05/01 14:38:52') AS \_month,  DATEPART(DAY, '2020/05/01 14:38:52') AS \_day,  DATEPART(HOUR, '2020/05/01 14:38:52') AS \_hour,  DATEPART(MINUTE, '2020/05/01 14:38:52') AS \_minute,  DATEPART(SECOND, '2020/05/01 14:38:52') AS \_second;    SELECT  DATENAME(YEAR, '2020/05/01 14:38:52') AS \_year,  DATENAME(MONTH, '2020/05/01 14:38:52') AS \_month,  DATENAME(DAY, '2020/05/01 14:38:52') AS \_day,  DATENAME(HOUR, '2020/05/01 14:38:52') AS \_hour,  DATENAME(MINUTE, '2020/05/01 14:38:52') AS \_minute,  DATENAME(SECOND, '2020/05/01 14:38:52') AS \_second; |

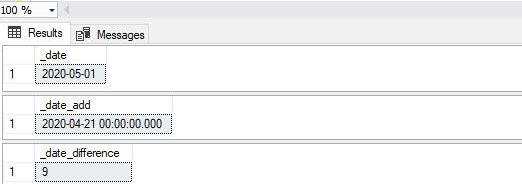


YEAR(…), MONTH(…), and DAY(…) return related parts of the given date.

DATEPART(date\_part, date) and DATENAME(date\_part, date) both return parts of the date. DATEPART returns them as integer values, while DATENAME returns them as strings. Please notice that you can define more than just a year, month or day. All date\_part values you can define are: year(yy, yyyy); quarter(qq, q); month(mm, m); dayofyear(dy, y); day(dd, d); week(wk, ww); weekday(dw); hour(hh); minute(mi, n); second(ss, s); millisecond(ms); microsecond(mcs); nanosecond(ns); TZoffset(tz); ISO\_WEEK(isowk, isoww).

The next 3 functions are used to create date or modify/combine dates. Let’s take a look at them.

|  |  |
| --- | --- |
| 1  2  3 | SELECT DATEFROMPARTS(2020,5,1) AS \_date;  SELECT DATEADD(DAY, -10, '2020-05-01') AS \_date\_add;  SELECT DATEDIFF(DAY, '2020-05-01', '2020-05-10') AS \_date\_difference; |



DATEFROMPARTS(year, month, day) takes a year, month and day as integer values and creates 1 date out of them.

DATEADD(date\_part, interval, date) takes 3 arguments and returns a date that is interval (date\_parts) number of given units (date\_part) distant from the given date (date).

DATEDIFF(date\_part, start\_date, end\_date) returns the number of units (date\_part) between end\_date and start\_date (end\_date – start\_date).

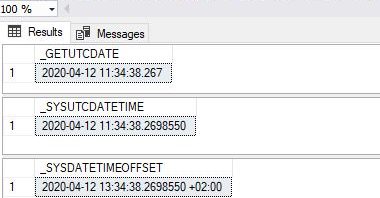
This is the end of my selection of the most commonly used SQL Server date and time functions. These functions should solve most of your “problems”. Still, there are some more, and we’ll cover them now.

Less frequently used SQL Server date and time functions

Let’s mention the less frequently used SQL Server date and time functions.

We’ll start with GETDATE() and SYSDATETIME() counterparts. Let’s take a look at the following statements:

|  |  |
| --- | --- |
| 1  2  3 | SELECT GETUTCDATE() \_GETUTCDATE;  SELECT SYSUTCDATETIME() \_SYSUTCDATETIME;  SELECT SYSDATETIMEOFFSET() AS \_SYSDATETIMEOFFSET; |



GETUTCDATE() acts the same as GETDATE() and CURRENT\_TIMESTAMP but it returns the UTC datetime value.

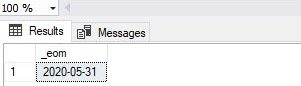
SYSUTCDATETIME() acts the same as SYSDATETIME() but it returns UTC values.

SYSDATETIMEOFFSET() is the same as SYSUTCDATETIME() but besides it also returns the time zone offset.

* **Note:** *This note is completely unrelated to date and time functions. Please notice that in the first 2 statements I haven’t used AS, while in 3rd I did it – with the same output (alias name had been used)*

Another interesting function is EOMONTH.

|  |  |
| --- | --- |
| 1 | SELECT EOMONTH('2020-05-01') AS \_eom; |



For the given date, it returns the last date in this month. This proves to be very useful in several situations.

I’ll list the remaining functions without giving examples:

* SWITCHOFFSET – Preserves the UTC value while changing the time zone to the one of a DATETIMEOFFSET value
* TODATETIMEOFFSET – Changes type from DATETIME2 into a DATETIMEOFFSET
* DATETIME2FROMPARTS – Creates and returns the DATETIME2 from the given date and time parts
* DATETIMEOFFSETFROMPARTS – Creates and returns the DATETIMEOFFSET from the given date and time parts
* TIMEFROMPARTS – Does the same as what DATEFROMPARTS does for the date, but for time. So, it creates a TIME from the given parts
* ISDATE – Is used to check if a given value is a valid datetime, date, or time, value

**Windowkey functions:**

* ROW\_NUMBER()
* RANK()
* DENSE\_RANK()
* NTILE()

In the SQL RANK functions, we use the OVER() clause to define a set of rows in the result set. We can also use SQL PARTITION BY clause to define a subset of data in a partition. You can also use Order by clause to sort the results in a descending or ascending order.

Before we explore these SQL RANK functions, let’s prepare sample data. In this sample data, we have exam results for three students in Maths, Science and English subjects.

We have the following sample data in the ExamResult table.

CREATE TABLE ExamResult

(StudentName VARCHAR(70),

Subject VARCHAR(20),

Marks INT

);

INSERT INTO ExamResult

VALUES ('Lily','Maths',65);

INSERT INTO ExamResult

VALUES ('Lily','Science',80);

INSERT INTO ExamResult

VALUES ('Lily','english', 70);

INSERT INTO ExamResult

VALUES ('Isabella', 'Maths',50);

INSERT INTO ExamResult

VALUES('Isabella','Science',70);

INSERT INTO ExamResult

VALUES('Isabella','english',90);

INSERT INTO ExamResult

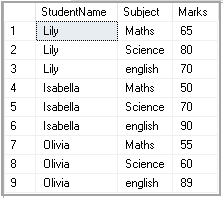
VALUES('Olivia','Maths',55);

INSERT INTO ExamResult

VALUES('Olivia','Science',60);

INSERT INTO ExamResult

VALUES('Olivia','english',89);



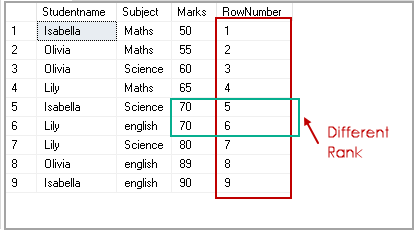
Let’s use each SQL Rank Functions in upcoming examples.

## ROW\_Number() function

We use ROW\_Number() SQL RANK function to get a unique sequential number for each row in the specified data. It gives the rank one for the first row and then increments the value by one for each row. We get different ranks for the row having similar values as well.

Execute the following query to get a rank for students as per their marks.

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT Studentname,         Subject,         Marks,         ROW\_NUMBER() OVER(ORDER BY Marks) RowNumber  FROM ExamResult; |



By default, it sorts the data in ascending order and starts assigning ranks for each row. In the above screenshot, we get ROW number 1 for marks 50.

We can specify descending order with Order By clause, and it changes the RANK accordingly.

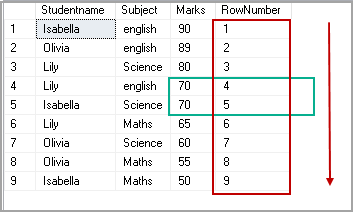
SELECT Studentname,

Subject,

Marks,

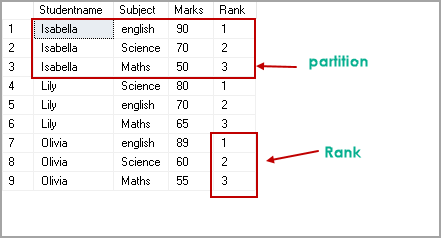
ROW\_NUMBER() OVER(ORDER BY Marks desc) RowNumber

FROM ExamResult;



## RANK() SQL RANK Function

We use RANK() SQL Rank function to specify rank for each row in the result set. We have student results for three subjects. We want to rank the result of students as per their marks in the subjects. For example, in the following screenshot, student Isabella got the highest marks in English subject and lowest marks in Maths subject. As per the marks, Isabella gets the first rank in English and 3rd place in Maths subject.



Execute the following query to get this result set. In this query, you can note the following things:

* We use PARTITION**BY Studentname** clause to perform calculations on each student group
* Each subset should get rank as per their Marks in descending order
* The result set uses Order By clause to sort results on Studentname and their rank

SELECT Studentname,

Subject,

Marks,

RANK() OVER(PARTITION BY Studentname ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY Studentname,

Rank;

Let’s execute the following query of SQL Rank function and look at the result set. In this query, we did not specify SQL PARTITION By clause to divide the data into a smaller subset. We use SQL Rank function with over clause on Marks clause ( in descending order) to get ranks for respective rows.

SELECT Studentname,

Subject,

Marks,

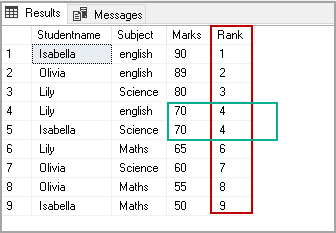
RANK() OVER(ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY Rank;

In the output, we can see each student get rank as per their marks irrespective of the specific subject. For example, the highest and lowest marks in the complete result set are 90 and 50 respectively. In the result set, the highest mark gets RANK 1, and the lowest mark gets RANK 9.

If two students get the same marks (in our example, ROW numbers 4 and 5), their ranks are also the same.



## DENSE\_RANK() SQL RANK function

We use DENSE\_RANK() function to specify a unique rank number within the partition as per the specified column value. It is similar to the Rank function with a small difference.

In the SQL RANK function DENSE\_RANK(), if we have duplicate values, SQL assigns different ranks to those rows as well. Ideally, we should get the same rank for duplicate or similar values.

Let’s execute the following query with the DENSE\_RANK() function.

SELECT Studentname,

Subject,

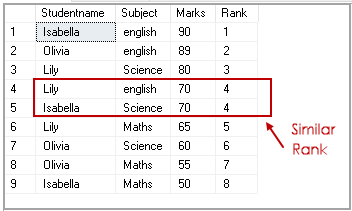
Marks,

DENSE\_RANK() OVER(ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY Rank;

In the output, you can see we have the same rank for both Lily and Isabella who scored 70 marks.



Let’s use DENSE\_RANK function in combination with the SQL PARTITION BY clause.

SELECT Studentname,

Subject,

Marks,

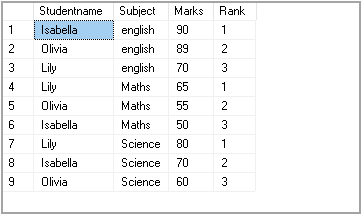
DENSE\_RANK() OVER(PARTITION BY Subject ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY Studentname,

Rank;

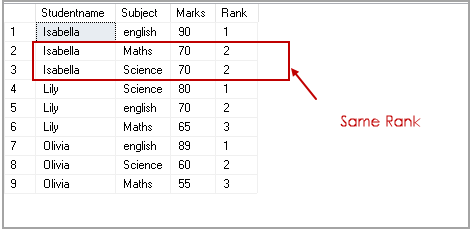
We do not have two students with similar marks; therefore result set similar to RANK Function in this case.



Let’s update the student mark with the following query and rerun the query.

|  |  |
| --- | --- |
| 1 | Update Examresult set Marks=70 where Studentname='Isabella' and Subject='Maths' |

We can see that in the student group, Isabella got similar marks in Maths and Science subjects. Rank is also the same for both subjects in this case.



Let’s see the difference between RANK() and DENSE\_RANK() SQL Rank function with the following query.

* Query 1

Select Studentname,

Subject,

Marks,

RANK() OVER(PARTITION BY StudentName ORDER BY Marks ) Rank

FROM ExamResult

ORDER BY Studentname,

Rank;

* Query 2

SELECT Studentname,

Subject,

Marks,

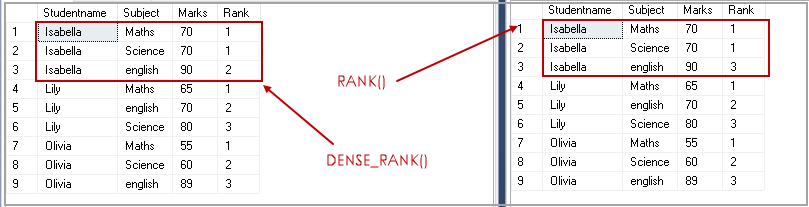
DENSE\_RANK() OVER(PARTITION BY StudentName ORDER BY Marks ) Rank

FROM ExamResult

ORDER BY Studentname,

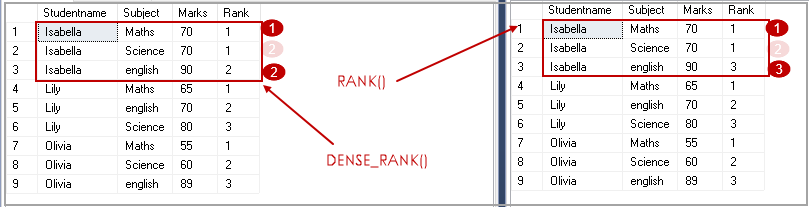
Rank;

In the output, you can see a gap in the rank function output within a partition. We do not have any gap in the DENSE\_RANK function.



In the following screenshot, you can see that Isabella has similar numbers in the two subjects. A rank function assigns rank 1 for similar values however, internally ignores rank two, and the next row gets rank three.

In the Dense\_Rank function, it maintains the rank and does not give any gap for the values.



## NTILE(N) SQL RANK function

We use the NTILE(N) function to distribute the number of rows in the specified (N) number of groups. Each row group gets its rank as per the specified condition. We need to specify the value for the desired number of groups.

In my example, we have nine records in the ExamResult table. The NTILE(2) shows that we require a group of two records in the result.

SELECT \*,

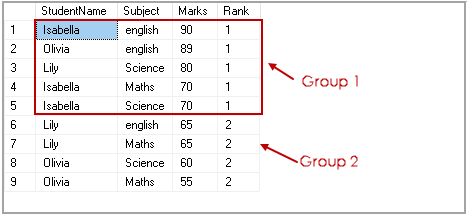
NTILE(2) OVER(

ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY rank;

In the output, we can see two groups. Group 1 contains five rows, and Group 2 contains four rows.



Similarly, NTILE(3) divides the number of rows of three groups having three records in each group.

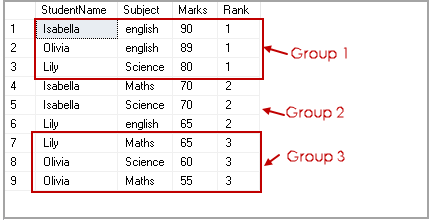
SELECT \*,

NTILE(3) OVER(

ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY rank;



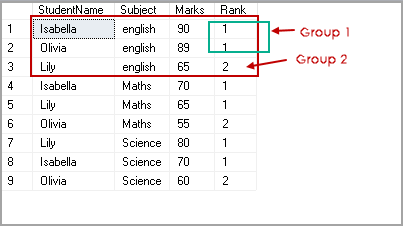
We can use SQL PARTITION BY clause to have more than one partition. In the following query, each partition on subjects is divided into two groups.

SELECT \*,

NTILE(2) OVER(PARTITION BY subject ORDER BY Marks DESC) Rank

FROM ExamResult

ORDER BY subject, rank;



## Practical usage of SQL RANK functions

We can use SQL RANK function to fetch specific rows from the data. Suppose we want to get the data of the students from ranks 1 to 3. In the following query, we use common table expressions(CTE) to get data using ROW\_NUMBER() function and later filtered the result from CTE to satisfy our condition.

WITH StudentRanks AS

(

SELECT \*, ROW\_NUMBER() OVER( ORDER BY Marks) AS Ranks

FROM ExamResult

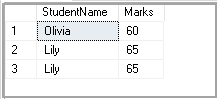
)

SELECT StudentName , Marks

FROM StudentRanks

WHERE Ranks >= 1 and Ranks <=3

ORDER BY Ranks;



We can use the OFFSET FETCH command starting from SQL Server 2012 to fetch a specific number of records.

WITH StudentRanks AS

(

SELECT \*, ROW\_NUMBER() OVER( ORDER BY Marks) AS Ranks

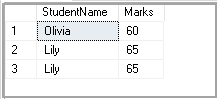
FROM ExamResult

)

SELECT StudentName , Marks

FROM StudentRanks

ORDER BY Ranks OFFSET 1 ROWS FETCH NEXT 3 ROWS ONLY;



## A quick summary of SQL RANK Functions

|  |  |
| --- | --- |
| ROW\_Number | It assigns the sequential rank number to each unique record. |
| RANK | It assigns the rank number to each row in a partition. It skips the number for similar values. |
| Dense\_RANK | It assigns the rank number to each row in a partition. It does not skip the number for similar values. |
| NTILE(N) | It divides the number of rows as per specified partition and assigns unique value in the partition. |

## index

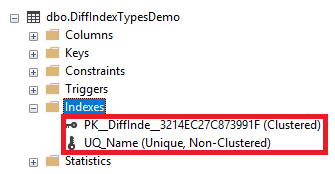
A Unique index is used to maintain the data integrity of the columns on which it is created, by ensuring that there are no duplicate values in the index key, and the table rows, on which that index is created. This ensures data will be unique based on the index key, depending on the characteristics of the data that is stored in the index key column or list of columns. If the Unique index key consists of one column, SQL Server will guarantee that each value in the index key is unique. On the other hand, if the Unique index key consists of multiple columns, each combination of values in that index key should be unique. You can define both the Clustered and Non-clustered indexes to be unique, as long as the data in these index keys are unique.

A Unique index will be created automatically when you define a PRIMARY KEY or UNIQUE KEY constraints on the specified columns. In all cases, creating a Unique index on the unique data, instead of creating a non-unique index on the same data, is highly recommended, as it will help the SQL Server Query Optimizer to generate the most efficient execution plan based on the additional useful information provided by that index.

Assume that we need to create the below table, using the CREATE TABLE T-SQL statement below, without specifying any CREATE INDEX statement, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE DiffIndexTypesDemo  ( ID INT IDENTITY (1,1) PRIMARY KEY,    Name Varchar(50) CONSTRAINT UQ\_Name UNIQUE,    ADDRESS NVARCHAR(MAX)  ) |

You will see that a Unique Clustered index will be created automatically on the ID column of that table, to enforce the PRIMARY KEY constraint, and a Unique Non-clustered index will be created automatically on the Name column to enforce the UNIQUE constraint, as shown below:



Take into consideration that the index that is created automatically to enforce any constraint cannot be dropped using a DROP INDEX T-SQL statement. If we try to drop the Unique index created previously to enforce the UNIQUE constraint using the DROP INDEX T-SQL statement below:

|  |  |
| --- | --- |
| 1 | DROP INDEX UQ\_Name ON DiffIndexTypesDemo |

the statement will fail, showing that we cannot explicitly drop any index that is created automatically to enforce a constraint, as shown in the error message below:

[](https://www.sqlshack.com/wp-content/uploads/2018/05/word-image-140.png)

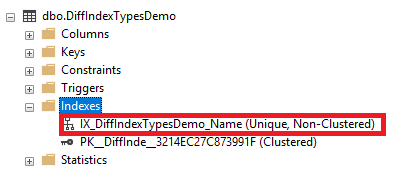
To drop that index, we should drop the constraint that created the index, using the ALTER TABLE…DROP CONSTRAINT T-SQL statement below:

|  |  |
| --- | --- |
| 1 | ALTER TABLE DiffIndexTypesDemo DROP CONSTRAINT UQ\_Name |

The Unique index can be also created manually, away from the constraint, by specifying the UNIQUE keyword in the Clustered or Non-Clustered index creation statement, as in the CREATE INDEX T-SQL statement below:

|  |  |
| --- | --- |
| 1 | CREATE UNIQUE NONCLUSTERED INDEX IX\_DiffIndexTypesDemo\_Name ON DiffIndexTypesDemo (NAME) |

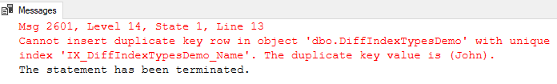
The previous CREATE INDEX statement can be used to create a Unique Non-Clustered index on the Name column, as shown below:



A Unique index is used to enforce the uniqueness of the index key values. For example, the previous index is used to make sure that no duplicate value for the Name column is available in that table. If we try to execute the below INSERT INTO statement that inserts two new records with the same Name values into that table:

|  |  |
| --- | --- |
| 1  2 | INSERT INTO DiffIndexTypesDemo VALUES ('John', 'Amman'),                                         ('John', 'Zarqa') |

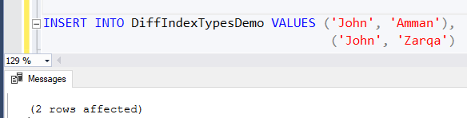
The statement will fail, showing that it is not allowed to insert duplicate values for the Name column, that is enforced by the created Unique index, providing the prevented duplicate values, as shown in the error message below:



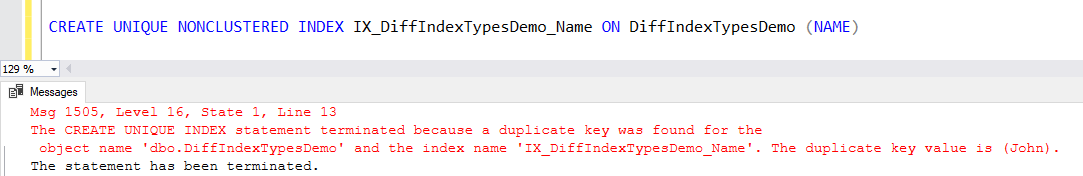
If we try to drop the Unique index, using the DROP INDEX T-SQL statement below:

|  |  |
| --- | --- |
| 1 | DROP INDEX IX\_DiffIndexTypesDemo\_Name ON DiffIndexTypesDemo |

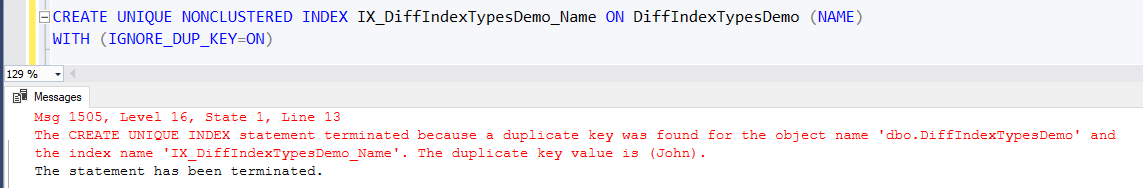
then execute the same INSERT INTO statement, you will see that the duplicate Name values will be inserted successfully, having no constraint or index that enforce the uniqueness of that column values, as shown clearly below:



Now, if we try to create the Unique index again on that table, the CREATE INDEX statement will fail, as the table already have duplicate values in the Name column as shown below:

[](https://www.sqlshack.com/wp-content/uploads/2018/05/word-image-144.png)

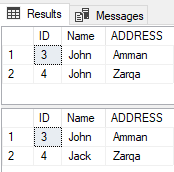
Neither will using the IGNORE\_DUP\_KEY index creation option will not work with the UNIQUE index. If we try to enable that option, while creating the Unique index, in order to ignore the existing duplicate values, the statement will fail again, showing that we cannot create a Unique index with duplicate index key values available in the table, as shown in the error message below:

[](https://www.sqlshack.com/wp-content/uploads/2018/05/word-image-145.png)

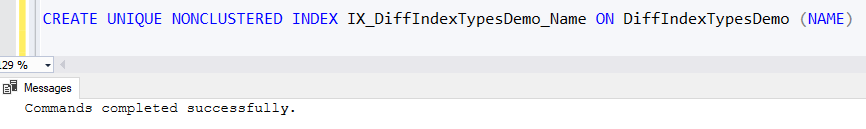
To be able to create the Unique index on the Name column, we should delete or update the duplicate values. In our case, we will update the second duplicate name using the UPDATE statement below:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM DiffIndexTypesDemo  UPDATE DiffIndexTypesDemo SET Name='Jack' where ID=4  SELECT \* FROM DiffIndexTypesDemo |

With the table rows before and after the UPDATE operation is shown below:



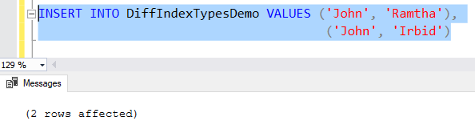
Trying to create the Unique index after resolving the duplicate issue, the Unique index will be created successfully as shown below:



We can include another column in the Unique index key to enforce the uniqueness of the combination of the two columns, rather than enforcing it on the Name column only. The below CREATE INDEX will be used to create a unique index that enforces the uniqueness of the ID and Name columns combination:

|  |  |
| --- | --- |
| 1 | CREATE UNIQUE NONCLUSTERED INDEX IX\_DiffIndexTypesDemo\_Name ON DiffIndexTypesDemo (ID,NAME) |

If we try to run the below INSERT INTO statement, that inserts two records with the same name, the records will be inserted successfully, as the ID column is IDENTITY column that will assign different values for each inserted row from the below:



## What is a CTE?

CTE is a 'temporary named result set'. In practice, a CTE is a result set that remains in memory for the scope of a single execution of a SELECT, INSERT, UPDATE, DELETE, or MERGE statement.

with test\_cte

as

(select customer.\*

from customer

inner join call on customer.id = call.customer\_id

)

select \* from test\_cte where city\_id > 3