**What is SET NOCOUNT ON**

**SET NOCOUNT** ON is a **set** statement which prevents the message which shows the number of rows affected by T-SQL query statements. This is used within **stored procedures** and triggers to avoid showing the affected rows message.

**What is GO**

The GO command is used to group SQL commands into batches which are sent to the server together. The commands included in the batch, that is, the set of commands since the last GO command or the start of the session, must be logically consistent. For example, you can't define a variable in one batch and then use it in another since the scope of the variable is limited to the batch in which it's defined.

**What is SET ANSI\_NULLS ON**

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. A SELECT statement that uses WHERE column\_name <> **NULL** returns zero rows even if there are nonnull values in column\_name

**What is SET ANSI\_PADDING { ON | OFF }**

Columns defined with **char**, **varchar**, **binary**, and **varbinary** data types have a defined size.

Trailing blanks in character values inserted into **varchar** columns are not trimmed. Trailing zeros in binary values inserted into **varbinary** columns are not trimmed. Values are not padded to the length of the column.

USE [Interview]

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[UpdateEmployeeSaleAmount] Script Date: 2/24/2018 4:41:40 PM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

ALTER PROCEDURE [dbo].[UpdateEmployeeSaleAmount]

@EmployeeSaleId int,

@EmployeeId int,

@SalesAmt decimal(16,2) = 0.00

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

BEGIN TRY

BEGIN TRANSACTION

UPDATE EmployeeSales Set SaleAmount = @SalesAmt

WHERE EmployeeId = @EmployeeId

AND EmployeeSales.EmployeeSaleId = @EmployeeSaleId

UPDATE Employee Set LastSalesAmtUpdate = GETDATE()

WHERE EmployeeId = @EmployeeId

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();

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

END CATCH

END

SQL Indexing and SQL Performance Part 2: Clustered and Non-Clustered

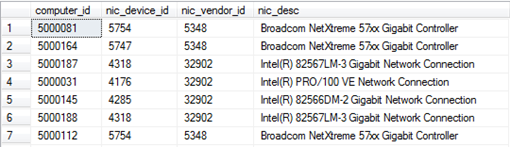
So, why do you need to index your tables?

Because without an index the SQL server has to scan the entire table to return the requested data. It is like the index page in a book. You check within the index for the keyword you want to learn about. From that point forward, you jump directly to the page where the content belongs, instead of scanning page by page for the material you want to read.

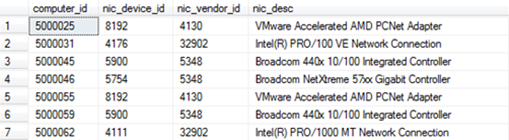
Similarly, a table’s index allows you to locate the data without the need to scan the entire table. You create instead indexes on one or more columns in a table, to aid in the query process. Consider the following example:

SELECT [computer\_id],[nic\_device\_id],[nic\_vendor\_id],[nic\_desc]

FROM [eXpress].[dbo].[nics]

[](https://www.giantstride.gr/wp-content/uploads/2013/08/image_24.png)

You are about to retrieve all computers with *computer\_id > 5100.* The SQL server will have to scan the entire table in order to return the results, without the presence of an index. An index on the *computer\_id* column will speed up this process, by sorting the column’s values.

[](https://www.giantstride.gr/wp-content/uploads/2013/08/image_thumb_25.png)

Now, if you want to return all data where *computer\_id > 5100*, the SQL server will know how to locate the first value greater than 5100. Why? Because the *computer\_id*column is sorted, eliminating the need to scan the entire table, thus improving performance.

SQL Index Types

There are two main index types: Clustered index and Non-Clustered index.

A clustered index alters the way that the rows are physically stored. When you create a clustered index on a column (or a number of columns), the SQL server sorts the table’s rows by that column(s).

It is like a dictionary, where all words are sorted in an alphabetical order. Note, that only one clustered index can be created per table. It alters the way the table is physically stored, it couldn’t be otherwise.

In the example below, all rows are sorted by *computer\_id,* as a clustered index on the computer\_id column has been created.

CREATE CLUSTERED INDEX [IX\_CLUSTERED\_COMPUTER\_ID]

ON [dbo].[nics] ([computer\_id] ASC)

A non-clustered index, on the other hand, does not alter the way the rows are stored in the table. Instead, it creates a completely different object within the table, that contains the column(s) selected for indexing and a pointer back to the table’s rows containing the data.

It is like an index in the last pages of a book. All keywords are sorted and contain a reference back to the appropriate page number. A non-clustered index on the computer\_id column, in the previous example, would look like the table below:

CREATE NONCLUSTERED INDEX [IX\_NONCLUSTERED\_COMPUTER\_ID]

ON [dbo].[nics] ([computer\_id] ASC)

|  |  |
| --- | --- |
| Computer\_id | Row Locator |
| 5000025 | 234 |
| 5000031 | 345 |
| 5000045 | 112 |
| 5000046 | 348 |
| 5000055 | 234 |
| 5000059 | 984 |

Note, that SQL server sorts the indexes efficiently by using a B-tree structure. This is a tree data structure that allows SQL Server to keep data sorted, to allow searches, sequential access, insertions and deletions, in a logarithmic amortized time. This methodology minimizes the number of pages accessed, in order to locate the desired index key, therefore resulting in an improved performance.

Relation between clustered and non-clustered indexes

As explained above, a non-clustered index contains a pointer back to the rowID ([RID](https://www.giantstride.gr/wp-content/uploads/2010/11/sql-indexing-and-performance-part-1-sql-storage-and-indexing)), of the table, in order to relate the index’s column with the rest of the columns in a row.

But this is not always the case:

If a clustered index already exists in the table, the non-clustered index uses the clustered index’s key as the row locator, instead of the [RID](https://www.giantstride.gr/wp-content/uploads/2010/11/sql-indexing-and-performance-part-1-sql-storage-and-indexing) reference.

In the example below, when a non-clustered index is created on the *computer\_id* column and a clustered index already exists on the *nic\_desc,* the non-clustered index would look like the table below.

|  |  |
| --- | --- |
| Computer\_id | Row Locator |
| 5000025 | VMware Accelerated AMD PCNet Adapter |
| 5000031 | Intel(R) PRO/100 VE Network Connection |
| 5000045 | Broadcom 440x 10/100 Integrated Controller |
| 5000046 | Broadcom NetXtreme 57xx Gigabit Controller |
| 5000055 | VMware Accelerated AMD PCNet Adapter |
| 5000059 | Broadcom 440x 10/100 Integrated Controller |

Index Benefits and Side Effects

A table without a clustered-index is called a *“heap table”.* A heap table has not its data sorted. The SQL server has to scan the entire table in order to locate the data, in a process called a *“scan”.*

In the case of a clustered index, the data are sorted on the key values (columns) of the index. The SQL server is now able to locate the data by navigating down from the root node, to the branch and finally to the leaf nodes of the B-tree structure of the index. This process called a *“seek”.* The later approach is much faster, when you want to filter or sort the data you want to retrieve.

A non-clustered index, on the other hand, is a completely different object in the table. It contains only a subset of the columns. It also contains a row locator looking back to the table’s rows, or to the clustered index’s key.

Because of its smaller size (subset of columns), a non-clustered index can fit more rows in an index page, therefore resulting to an improved I/O performance. Furthermore a non-clustered index can be allocated to a different FileGroup, which can utilize a different physical storage in order to improve performance even more.

The side effects of indexes are related to the cost of *INSERT*, *UPDATE, MERGE* and *DELETE*statements. Such statements can take longer to execute, in the presence of indexes, as it alters the data in the table, thus to the indexes too.

Imagine the situation of an *INSERT* statement. It has to add new rows in a table with a clustered index. In such case the table rows may need to be re-positioned! Remember…? The clustered index needs to order the data pages themselves! This will cause overhead.

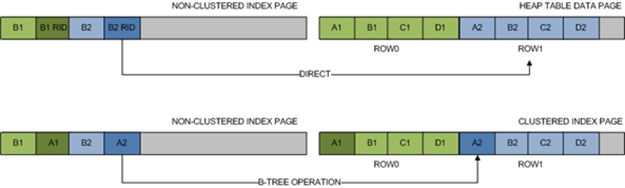
So, it is crucial to take into account the overhead of *INSERT*, *UPDATE* and *DELETE* statements before designing your indexing strategy. Although there is an overhead to the above statements, you have to take into account, that many times, an *UPDATE* or *DELETE* statement will execute in a subset of data. This subset can be defined by a WHERE clause, where indexing may outweigh the additional cost of index updates, because the SQL server will have to find the data before updating them.

As explained above, a non-clustered index includes the clustered index’s key as its row locator, in the presence of a clustered index in the table.

This comes with a cost and a benefit:

The cost has to do with the non-clustered index *bookmark lookup*. What if a query has to return more columns that the ones hosted in the index itself? In the case of a *HEAP* table, the SQL server would have to check the RID of the non-clustered index, in order to navigate directly to the row, where the rest of the columns belong.

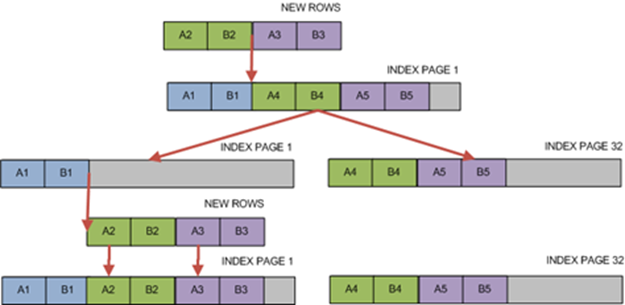
In the case of a clustered index, the SQL server would have to check the row locator of the non-clustered index, in order to do an additional navigation to the B-tree structure of the clustered index, to retrieve the desired row. You see, the row locator does not contain the RID, but the clustered-index key.

[](https://www.giantstride.gr/wp-content/uploads/2013/08/image_thumb_26.png)

On the other hand, there is a benefit. It has to do with the clustered index updates. Imagine the following situation: Two new rows with index key values of A2 and A3 have to be added in the clustered index below.

[image_thumb_27](https://www.giantstride.gr/wp-content/uploads/2013/08/image_thumb_27.png)

Because this is a clustered index page, its physical structure has to be reallocated in order to fit A2 and A3 between A1 and A4. It has to maintain index’s order. Since there is no free space in the index page to accommodate these changes,  a *page split* will occur. Now, there is enough space to fit A2 and A3 between A1 and A4.

[](https://www.giantstride.gr/wp-content/uploads/2013/08/image_thumb_28.png)

The goal achieved and the order maintained within the index. But imagine what would have happened if the non-clustered index was looking at the RID, instead of the clustered index’s key? It would have to change its row locators to reflect the changes. This could have been a huge performance hit! Especially, in the case of large clustered indexes.

Instead of the RID, the row locators now point at the clustered index key. Meaning, that there is no longer needed to change its values. This is quite a benefit if you think of the large clustered indexes, that are usually maintained in many tables.

Clustered indexes VS non-clustered indexes

|  |  |  |
| --- | --- | --- |
|  | CLUSTERED | NON-CLUSTERED |
| PROS | * Fast to return large range of data * Fast for presorted results | * Wide keys do not reflect on other indexes * Frequently updated key columns do not reflect on other indexes * Can be assigned on different FileGroup * Many non-clustered indexes per table * Smaller size than clustered indexes due to column subsets |
| CONS | * Frequently updated key columns reflect on non-clustered indexes * Wide keys increase the size of the non-clustered indexes * Only one clustered index per table | * Generally slower than clustered indexes due to bookmark lookup (except for covering indexes). * Not recommended for returning large data sets (except for covering indexes). |

**https://d10lpsik1i8c69.cloudfront.net/graphics/mail.png**

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

**BUILD IN STRING FUNCTIONS**

LTRIM(Character\_Expression) - Removes blanks on the left handside of the given character expression.  
  
**Example**: Removing the 3 white spaces on the left hand side of the '   Hello' string using LTRIM() function.  
Select LTRIM('   Hello')  
**Output**: Hello  
  
RTRIM(Character\_Expression) - Removes blanks on the right hand side of the given character expression.

LOWER(Character\_Expression) - Converts all the characters in the given Character\_Expression, to lowercase letters.  
  
**Example**: Select LOWER('CONVERT This String Into Lower Case')  
**Output**: convert this string into lower case  
  
UPPER(Character\_Expression) - Converts all the characters in the given Character\_Expression, to uppercase letters.  
**Example**: Select UPPER('CONVERT This String Into upper Case')  
**Output**: CONVERT THIS STRING INTO UPPER CASE  
  
REVERSE('Any\_String\_Expression') - Reverses all the characters in the given string expression.  
**Example**: Select REVERSE('ABCDEFGHIJKLMNOPQRSTUVWXYZ')  
**Output**: ZYXWVUTSRQPONMLKJIHGFEDCBA  
  
LEN(String\_Expression) - Returns the count of total characters, in the given string expression, excluding the blanks at the end of the expression.

**LEFT**(Character\_Expression, Integer\_Expression) - Returns the specified number of characters from the left hand side of the given character expression.  
  
**Example**: Select LEFT('ABCDE', 3)  
**Output**: ABC  
  
**RIGHT**(Character\_Expression, Integer\_Expression) - Returns the specified number of characters from the right hand side of the given character expression.  
  
**Example**: Select RIGHT('ABCDE', 3)  
**Output**: CDE  
  
**CHARINDEX**('Expression\_To\_Find', 'Expression\_To\_Search', 'Start\_Location') - Returns the starting position of the specified expression in a character string. Start\_Location parameter is optional.  
  
**Example**: In this example, we get the starting position of '@' character in the email string 'sara@aaa.com'.   
Select CHARINDEX('@','sara@aaa.com',1)  
**Output**: 5  
  
**SUBSTRING**('Expression', 'Start', 'Length') - As the name, suggests, this function returns substring (part of the string), from the given expression. You specify the starting location using the 'start' parameter and the number of characters in the substring using 'Length' parameter. All the 3 parameters are mandatory.  
  
**Example**: Display just the domain part of the given email 'John@bbb.com'.  
Select SUBSTRING('John@bbb.com',6, 7)  
**Output**: bbb.com

-------------------------------------------------------------------------------------------------------------------------------------------

### DateTime functions in SQL Server

There are several built-in DateTime functions available in SQL Server. All the following functions can be used to get the current system date and time, where you have sql server installed.

|  |  |  |
| --- | --- | --- |
| **Function** | **Date Time Format** | **Description** |
| GETDATE() | 2012-08-31 20:15:04.543 | Commonly used function |
| CURRENT\_TIMESTAMP | 2012-08-31 20:15:04.543 | ANSI SQL equivalent to GETDATE |
| SYSDATETIME() | 2012-08-31 20:15:04.5380028 | More fractional seconds precision |
| SYSDATETIMEOFFSET() | 2012-08-31 20:15:04.5380028 + 01:00 | More fractional seconds precision + Time zone offset |
| GETUTCDATE() | 2012-08-31 19:15:04.543 | UTC Date and Time |
| SYSUTCDATETIME() | 2012-08-31 19:15:04.5380028 | UTC Date and Time, with More fractional seconds precision |
|  |  |  |

### ISDATE() - Checks if the given value, is a valid date, time, or datetime. Returns 1 for success, 0 for failure. Examples: Select ISDATE('PRAGIM') -- returns 0 Select ISDATE(Getdate()) -- returns 1 Select ISDATE('2012-08-31 21:02:04.167') -- returns 1

### Day() - Returns the 'Day number of the Month' of the given date Examples: Select DAY(GETDATE()) -- Returns the day number of the month, based on current system datetime. Select DAY('01/31/2012') -- Returns 31 Month() - Returns the 'Month number of the year' of the given date Examples: Select Month(GETDATE()) -- Returns the Month number of the year, based on the current system date and time Select Month('01/31/2012') -- Returns 1 Year() - Returns the 'Year number' of the given date Examples: Select Year(GETDATE()) -- Returns the year number, based on the current system date Select Year('01/31/2012') -- Returns 2012

### Select DATENAME(Day, '2012-09-30 12:43:46.837') -- Returns 30 Select DATENAME(WEEKDAY, '2012-09-30 12:43:46.837') -- Returns Sunday Select DATENAME(MONTH, '2012-09-30 12:43:46.837') -- Returns September

**Similarities between RANK, DENSE\_RANK and ROW\_NUMBER functions**

* Returns an increasing integer value starting at 1 based on the ordering of rows imposed by the ORDER BY clause (if there are no ties)
* ORDER BY clause is required
* PARTITION BY clause is optional
* When the data is partitioned, the integer value is reset to 1 when the partition changes

**Difference between RANK, DENSE\_RANK and ROW\_NUMBER functions**

* **ROW\_NUMBER :** Returns an increasing unique number for each row starting at 1, even if there are duplicates.
* **RANK :**Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows, but the next row after the duplicate rows will have the rank it would have been assigned if there had been no duplicates. So RANK function skips rankings if there are duplicates.
* **DENSE\_RANK :**Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows but the DENSE\_RANK function will not skip any ranks. This means the next row after the duplicate rows will have the next rank in the sequence.

When to Use Temporary Tables?

Below are the scenarios where we can use temporary tables:

When we are doing large number of row manipulation in stored procedures.

This is useful to replace the cursor. We can store the result set data into a temp table, then we can manipulate the data from there.

When we are having a complex join operation.

Points to Remember Before Using Temporary Tables

Temporary table created on tempdb of SQL Server. This is a separate database. So, this is an additional overhead and can causes performance issues.

Number of rows and columns need to be as minimum as needed.

Tables need to be deleted when they are done with their work.

Alternative Approach: Table Variable

Alternative of Temporary table is the Table variable which can do all kinds of operations that we can perform in Temp table. Below is the syntax for using Table variable.

Declare @TempTableVariable TABLE(

UserID int,

UserName varchar(50),

UserAddress varchar(150))

The below scripts are used to insert and read the records for Tablevariables:

Hide Copy Code

insert into @TempTableVariable values ( 1, 'Abhijit','India');

Now select records from that tablevariable:

select \* from @TempTableVariable

When to Use Table Variable Over Temp Table

Tablevariable is always useful for less data. If the result set returns a large number of records, we need to go for temp table.